NASA Contractor Report 189604

SIMULATOR FOR CONCURRENT PROCESSING DATA FLOW ARCHITECTURES

Mahyar R. Malekpour, John W. Stoughton, and Roland R. Mielke

OLD DOMINION UNIVERSITY RESEARCH FOUNDATION
Norfolk, Virginia

Grant NCC1-136
March 1992

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

PREFACE

This report contains essentially the thesis written by Mahyar R. Malekpour entitled "Simulator for Concurrent Processing Data Flow Architectures" in fulfillment of the Masters Degree at Old Dominion University. The use of brand names in this report is for completeness and does not imply NASA endorsement.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

FIGURES PAGE

LIST OF SYMBOLS

SYMBOL	DESCRIPTION
ADM	Advanced Development Model
AMG	Algorithm Marked Graph
AMOS	ATAMM Multicomputer Operating System
ATAMM	Algorithm to Architecture Mapping Model
C_i	ith circuit in the CMG
CMG	Computational Marked Graph
DP	Data processed
DR	Data read
E	FDT file event: waiting for the channel in order to read
F	FDT file event: reading
FDT	Fire Data Time
FU	Functional Unit
I	FDT file event: processing
IBM	International Business Machines
ID	Identification
IE	Input buffer empty
IF	Input buffer full
I/O	Input/Output
$M(C_i)$	Number of tokens in C_i
N	Number of nodes in the AMG
NMG	Node Marked Graph
O	FDT file event: testing

OE	Output buffer empty
OF	Output buffer full
P	FDT file event: waiting for the channel in order to write
P_i	ith path between source and sink in AMG
PR	Process ready
Q	FDT file event: returning FU ID to the resource queue
R	FDT file event: idle
S	FDT file event: writing
T	FDT file event: waiting for the channel in order to return the FU ID to the resource queue; or a time interval
T1	Time interval for one stay in the idle state of AMOS
T2	Time interval for one stay in the examine state of AMOS
T3	Time interval of the execute state of AMOS
T4	Time interval of the test state of AMOS
T5	Time interval of the update state of AMOS
TB	Broadcast time
TBI	Time Between Inputs
TBIO	Time Between Input and Output (Graph Latency)
$TBIO_{LB}$	Lower bound limit of TBIO
TBO	Time Between Outputs
TBO_{LB}	Lower bound limit of TBO
TBO_{min}	Minimum TBO due to overhead requirements
TCE	Total Computing Effort
$T(C_i)$	Sum of transition times in C_i
T_e	Evaluation time interval
TE	Time to execute the algorithm operation belonging to the critical node

TG	Channel grab time
Simplex	A single copy of a graph
Duplex	Two copies of a graph
TMR	Triple Modular Redundancy, three copies of a graph
$T(P_i)$	Sum of transitions times in P_i
STP	Set up file extension
CTL	Control file extension
GPH	Graph file extension
FDT	Fire, data, time file extension

CHAPTER ONE

Introduction

1.1 Overview

As the use of computers affects increasingly broader segments of the world, many of the problems to which people apply computers grow continually larger and more complex. Demands for faster and larger computer systems increase steadily and outpace the recent advances in technology. Computer architects have followed two general approaches in response to this demand. The first uses exotic technology in a fairly conventional serial computer architecture. The second approach exploits the parallelism inherent in many problems. The parallel approach requires a system with multiple processors working concurrently on the same algorithm. Due to inherent concurrency in applications such as real-time signal processing and control systems, a special model is needed to describe the system behavior and predict its performance for real-time application.

Strategies for control of computations on multicomputer architectures can be classified broadly as control-flow, demand driven, and data-driven [1]. In control flow computers, explicit flows of control cause the execution of instructions. In demand-driven architectures, the execution of operations are triggered by the requirements for outputs. In data-driven architectures (also known as data-flow computers), the availability of operands triggers the execution of operations.

The data-flow concept has already attracted the attention of many researchers [2]. A number of decentralized data-flow architectures have been developed, motivated mainly by the desire to improve performance through the use of concurrency [3]. However, only a few researchers have tried to develop a theoretical model for evaluating

computation in a data-driven architecture [4], [5]. These models do not appear to be adequate in addressing the complex issues of repeated execution of algorithms. There is a need for a simple, but effective, model for real-time, data-driven computations in order to investigate the relative merits of different algorithm decompositions and implementation strategies in a hardware independent context. Ongoing research efforts at Old Dominion University have lead to the development of a new marked graph model for describing data and control flow associated with the execution of algorithms in real-time data flow architectures [6], [7]. The model is identified by the acronym ATAMM which stands for Algorithm To Architecture Mapping Model [8], [1]. The model was designed by Stoughton and Mielke in conjunction with NASA Langley Research Center, in order to describe the control, communication, and scheduling issues not included in other models [8]. The architecture is assumed to be a homogeneous multicomputer data flow architecture consisting of two to twenty identical computers or functional units each having a capability for processing, communication, and memory. The algorithms are assumed to be decision free and require large computations, i.e., large-grained, which include such computations as matrix addition, multiplication, etc.. The granularity level of the algorithm decomposition is set high to keep the relative communication overhead small.

The Algorithm to Architecture Mapping Model (ATAMM) is a new Petri net based model capable of describing the execution of large-grained algorithms on data-flow architectures. In the data-flow model of computation, operations proceed on the availability of data rather than the action of a program counter as in the von Neuman model of computers. Large- grained means that the time required to execute data by an algorithm operation is much greater than the time to transfer data between the operations.

The ATAMM model provides a description of the data and control flow necessary to specify the criteria for predictable execution of an algorithm by a data-flow architecture. The ATAMM model also provides the means to investigate different

algorithm decompositions without having to consider the hardware. Once the intended hardware is selected, the model can be used to match the algorithm requirements with the hardware capability in order to achieve optimum performance.

1.2 Research Objective

The objective of this research is to develop a software simulator capable of simulating execution of a graph on a given system under the ATAMM rules. The purpose of the simulator is to empower a study of behavior, performance, and reliability of a multicomputer data flow system without having to build a hardware prototype. This simulator is able to assist with the development of ATAMM-based architectures and the investigation of theories concerning the ATAMM model. The simulator is to be user-friendly and flexible to permit examining different attributes of a generic system. Evaluation of the simulator is conducted through several case studies.

The simulator provides the means to identify an architecture by specifying different parameters of the system in order to evaluate the periodic execution of an algorithm on a given hardware. Architecture parameters include such variables as graph-node execution time, communication latency, and memory read-write latencies. The simulator is capable of detecting and recovering from faults, and also provides the means to obtain performance measurements. The performance measurements indicate the graph latency, throughput, concurrency, and resource utilization attained by the system. Results of execution of an algorithm by the simulator are comparable to the results of the ADM system. In order to ease and facilitate user interactions, this user-friendly software is developed within a window environment.

1.3 Organization

A brief description of the ATAMM model and related performance issues are presented in Chapter Two. The ATAMM model and its application is presented in

Section 2.2. Section 2.3 is a brief introduction of the Advanced Development Model (ADM) system. In Section 2.4, the ATAMM Multicomputer Operating System (AMOS) for the ADM system is presented as an example implementation of ATAMM. The major components of the AMOS are identified and AMOS operation is explained using a state diagram description and an overhead model associated with AMOS operation is considered. Also, an approach that extends ATAMM to the modeling of a fault tolerant system is presented in Section 2.5. Different classes of faults and fault tolerance strategies in AMOS and the ADM system are discussed.

The development of software for design and simulation of an ATAMM based system is presented in Chapter Three. Section 3.2 is a brief introduction of object-oriented methodology. A brief description of system requirements, the programming environment and the language used in the development of the ATAMM Simulator is presented in Section 3.3. Objects and relationships among them are introduced in Section 3.5. The Simulator-Kernel, Processors, Network, Graph, Graph-Manager, and PC objects are discussed in Sections 3.6 through 3.11, respectively. Inputs and outputs of the simulator and the format of the input and output files are discussed in Section 3.12.

Experimental results are described in Chapter Four and provide a demonstration of the software developed in Chapter Three. Real algorithms are chosen for case studies to prove the practical applicability of the ATAMM Simulator. The last algorithm considered for the case studies is a three-node chain graph. The simulated results of this graph are compared with that of the ADM system and are presented in Section 4.6. Finally, some observations and experimental findings are addressed in Section 4.7.

A summary and topics for future research are stated in Chapter Five.

CHAPTER TWO

Background

2.1 Introduction

The purpose of the ATAMM Simulator is to model the behavior of multicomputer architectures based on the ATAMM model. The ATAMM model for describing the data and control flow associated with a certain class of algorithms and distributed-processing systems is presented in this chapter. A brief description of the ATAMM model and its application is presented in Section 2.2. Section 2.3 is a brief introduction of the Advanced Development Model (ADM) system. The ADM is an example of an ATAMM implementation on the VHSIC hardware. In Section 2.4, the major components of the ATAMM Multicomputer Operating System (AMOS) are identified and AMOS operation is explained using a state diagram description. Different classes of faults and fault tolerance strategies in AMOS and the ADM system are discussed in Section 2.5.

2.2 ATAMM Model

Multicomputers are increasingly being used for real-time applications such as aerospace or nuclear power plants [9]. In a typical real-time system, a number of sensors provide input data to a computer which then analyzes the input data by some predefined algorithms. This information is used to send output signals to actuators or displays. A few characteristics required of such computers are repeated execution of the same algorithm, highly predictable and reliable performance, and hard deadlines for outputs.

The ATAMM model is based on a special class of timed Petri nets. Petri nets are a tool for the study of systems with discrete events. A Petri net is a special kind of directed graph capable of describing data and control flow of a system [10]. Petri nets

serve as both a graphical and mathematical tool. The reader is expected to be familiar with Petri net theory so a detailed discussion pertaining to the topic will not be provided. The reader unfamiliar with Petri nets may refer to [10] for a discussion of Petri net theory.

An important subclass of Petri net is the marked graph. In a marked graph every edge is an input to only one transition and an output of exactly one transition. In other words, each edge has exactly one input and one output. This restriction eliminates the possible conflict of having one place as input to more than one transition. Marked graphs can be used to model the processing of decision-free algorithms [5]. An example of a marked graph is presented in Figure 2.1. Circles represent nodes (transitions) and line

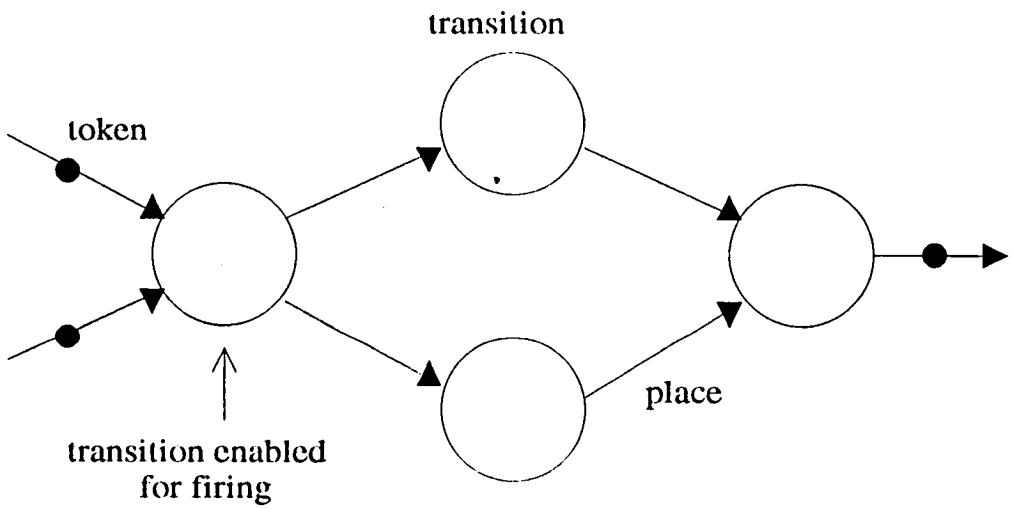

Figure 2.1. Partial Marked Graph.

segments represent edges (places). Tokens representing the availability of signals or data are indicated by black dots on the edges. A node is "enabled" for "firing" by the presence of tokens on all of its input edges. The node "fires" by encumbering all input tokens, delaying for some time interval, and depositing one token on each output edge.

The ATAMM model provides the analytical means to integrate the algorithm data flow with the data flow architecture [11]. The ATAMM model consists of three Petri net

marked graphs which incorporate general specifications of communication and processing associated with each computational event in a data flow architecture. The algorithm marked graph (AMG), the node marked graph (NMG), and the computational marked graph (CMG) constitute the three main components of the ATAMM model. A flow diagram portraying the ATAMM modeling integration is presented in Figure 2.2.

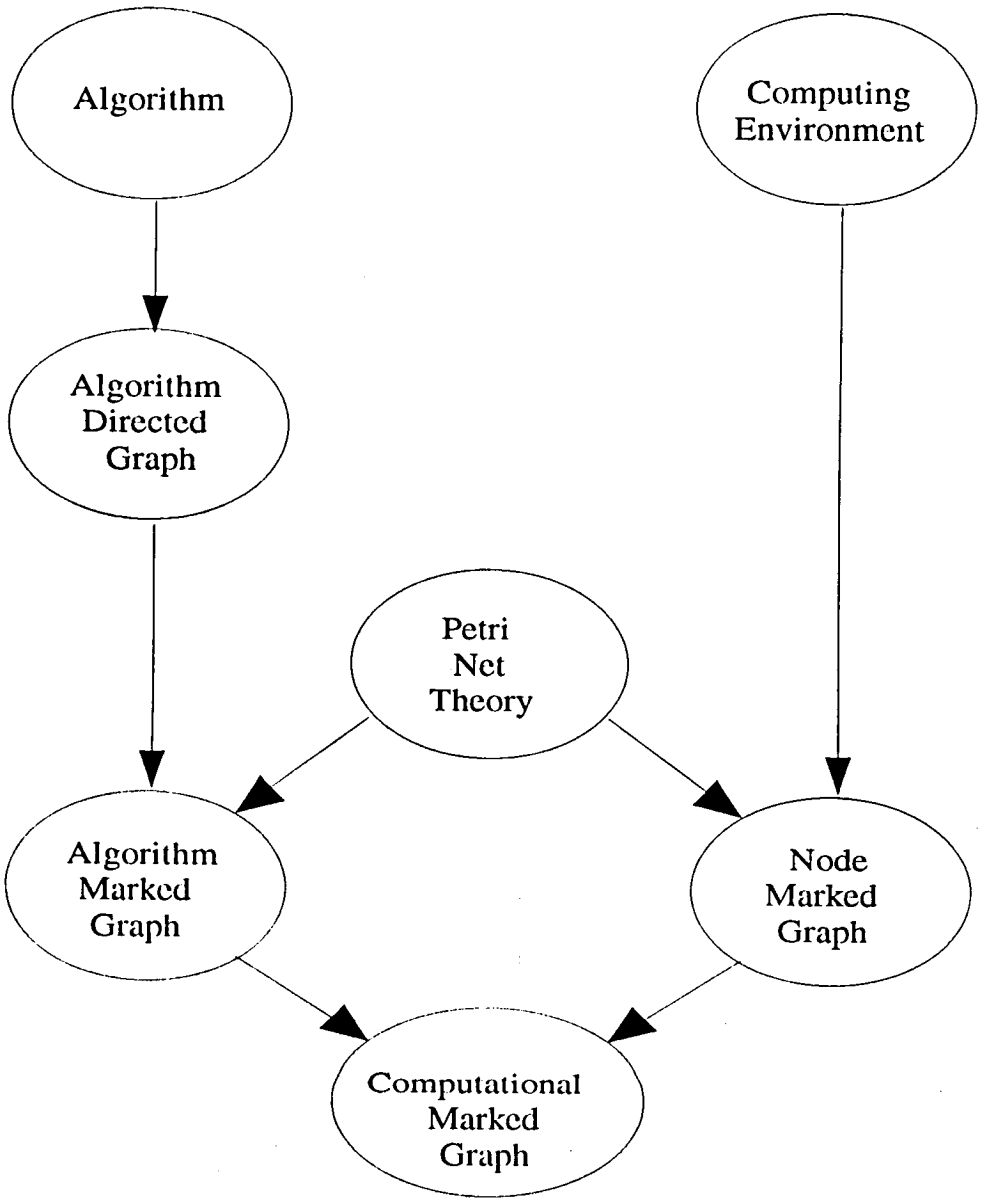

Figure 2.2. ATAMM Model Components.

Given an algorithm decomposition, the primitive operations and their data dependencies are described by the algorithm marked graph (AMG). While the nodes (circles) represent different tasks, the edges (line segments) represent data dependence as well as data containers. Tokens are used to indicate the presence of data on the edges. Squares are used to indicate sources and sinks of algorithm marked graphs. An example AMG is provided in Figure 2.3.

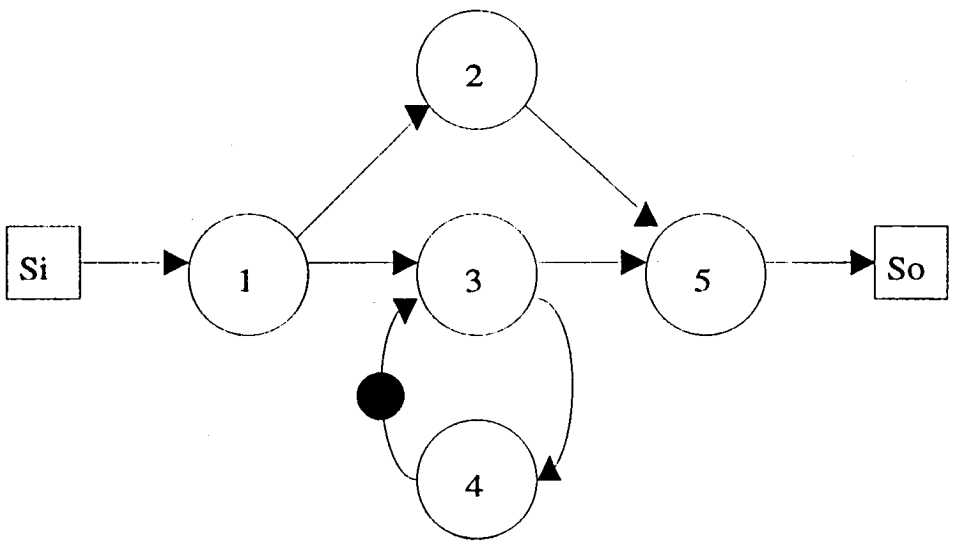

Figure 2.3. Example AMG.

It is desirable to analyze the performance of a particular algorithm graph without actually running it on any hardware. It is also important to be able to develop control parameters to ensure the orderly computation of the tasks. These problems are resolved in the graph theoretic context by the Node Marked Graph (NMG). The NMG is also a marked graph. Given some computing environment assumptions, the NMG specifies the functional unit activities which must occur in order to execute a primitive operation represented by an AMG node. One assumption is that the computing environment will contain global memory for storage of data associated with each AMG edge. The global memory may be centralized or distributed among the functional units [11]. Each

functional unit contains local memory for the storage of data and the code to execute any primitive operation of the AMG. A functional unit must read data from global memory into its local data container, process the data, and write the data back to global memory for access by other functional units. To ensure safety of data, a functional unit is not able to start processing a task until output data for that operation has been consumed. This is implemented by introduction of backward control edges in an AMG from the successor to the predecessor node. An NMG describing these activities is displayed in Figure 2.4.

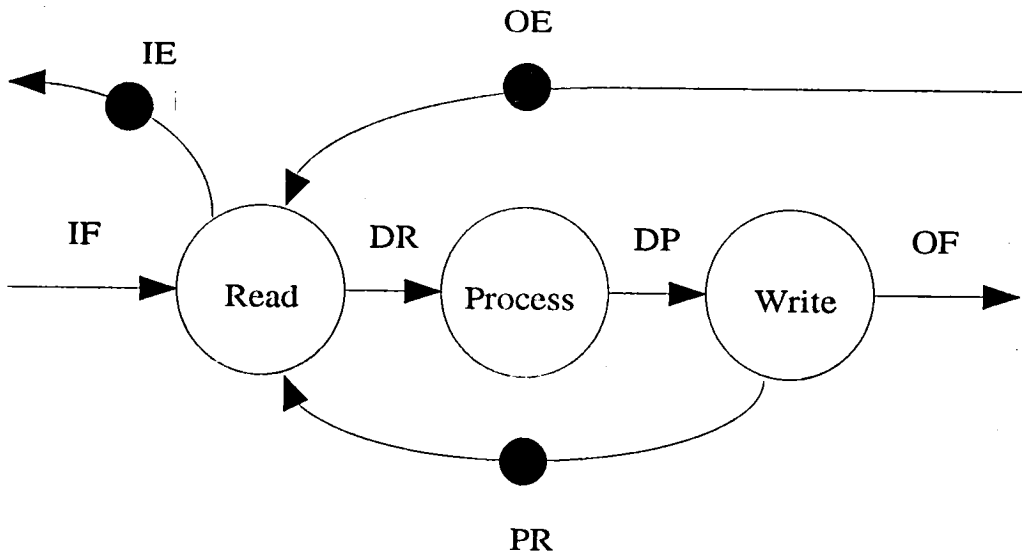

Figure 2.4. NMG Description Computing Activity.

A token at the labeled edges indicates the following:

IF	Input Buffer Full
IE	Input Buffer Empty
OF	Output Buffer Full
OE	Output Buffer Empty
DR	Data Read
DP	Data Processed
PR	Process Ready

The NMG of Figure 2.4 is a detailed specification of not only the activities to be performed by a functional unit but also the conditions which "enable" those operations. The read node is "enabled" when it is ready, input is available, and the output has been read by the successor operation. To "fire" the node, a functional unit is assumed to be available to undertake these activities. The functional unit assigned to the read transition will not be available until after completion of processing the AMG node and writing the data. This is indicated by appearance of tokens on the output edges of the AMG node.

The two modeling steps of ATAMM discussed so far have specified data flow with the AMG, and the functional unit activities and control flow required of each AMG node. The CMG is a marked graph which incorporates the AMG and NMG specifications into one graph. Thus, the CMG displays the data and control flow necessary to implement a decomposed algorithm on a multiprocessor data flow architecture [11]. The CMG is constructed from the AMG by replacing every transition by the corresponding NMG. The source and sink of the AMG are represented the same way in the CMG. AMG edges are replaced by edge pairs, a forward directed edge for data flow and a backward directed edge for control flow. The resulting CMG is shown in Figure 2.5.

The CMG of Figure 2.5 has certain characteristics that should be briefly mentioned. Execution of the CMG results in live, reachable, safe, deadlock free and consistent behavior. Liveness indicates that every transition of the graph can be fired from the initial marking [8]. Reachability implies that an output will be produced for every input. The CMG is safe because the backward control edges prevent data from being overwritten. The backward control edge prevents the enabling of a primitive operation until previous output data are consumed. The CMG is also deadlock-free because once assigned to a primitive operation, a functional unit will always be able to complete execution. Consistency implies that the CMG will periodically produce output

when input is applied periodically [8]. This also means that primitive operations also are executed periodically.

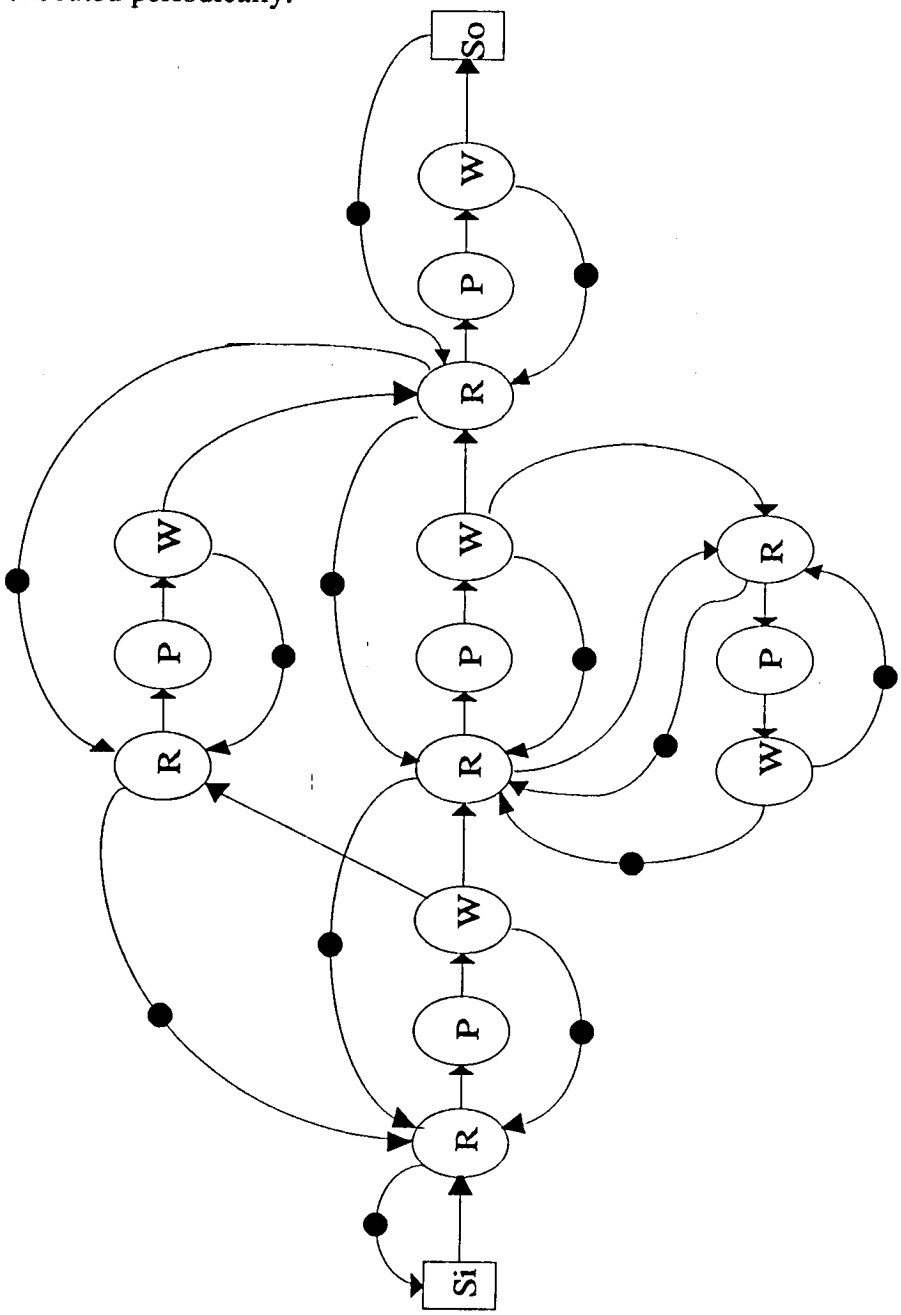

Figure 2.5. Example CMG.

Two types of concurrency are possible when executing an algorithm decomposition as specified by the CMG. First, primitive operations without data

dependency may be simultaneously performed on the same data set. This is referred to as parallel concurrency and provides parallelism on a single data set [12]. It is the result of inherent parallelism in the algorithm. However, the amount of parallel concurrency depends on the number of parallel paths in the algorithm decomposition and the number of available functional units. Second, as with any data flow computer, new data sets are accepted for execution before the completion of previous data set computations. This simultaneous processing of different data sets is referred to as pipeline concurrency [12]. This type of concurrency has a direct effect on throughput. The amount of pipeline concurrency depends on the number of available functional units as well as the structure of the AMG.

The AMG and CMG of a given algorithm decomposition can be used to calculate performance measurements. Two important performance measurements are the time between input and output (TBIO) and the time between outputs (TBO). TBIO is directly related to graph latency which indicates the amount of parallel concurrency attained. TBO is associated with throughput and therefore reflects the amount of pipeline concurrency achieved.

Lower bounds for TBIO and TBO can be calculated using the CMG. The lower bound, $TBIO_{LB}$, is determined from the CMG by determining the longest path between the input source and the output sink. More formally, let P_i be the ith directed path in the CMG and $T(P_i)$ be the total path time associated with P_i. $TBIO_{LB}$ is then defined as

$$TBIO_{LB} = Max(T(P_i)), \qquad (2.1)$$

where the maximum is taken over all paths in the CMG [8]. A proof of this theorem can be found in [1] and is based on critical path theory.

TBO_{LB} is a parameter indicating how quickly primitive operations can be repeated periodically. Let C_i be the i^{th} directed circuit in the CMG and $T(C_i)$ denote the

total path time associated with C_i. Also, let $M(C_i)$ denote the number of tokens contained in C_i. Then, TBO_{LB} is defined as

$$TBO_{LB} = Max(T(C_i)/M(C_i)), \qquad (2.2)$$

where the maximum is taken over all circuits in the CMG [8]. TBO_{LB} is thus the largest time per token of all CMG circuits. The CMG circuits which determine TBO_{LB} are called critical circuits. A proof of Equation 2.2 can be found in [1] and is based on the maximum node firing rate of marked graphs.

Knowledge of TBO_{LB} is important because it determines the minimum injection interval of graph input. Data may temporarily be accepted within a time interval shorter than TBO_{LB} but at the cost of increased graph latency (TBIO will increase). However, it is important in real-time applications to have low graph latency as well as high throughput. The ATAMM model provides the means to match the algorithm requirements with resource availability for optimum performance and to establish the criteria for predictable performance. Predictable performance is attained by maintaining an input injection rate within the range determined by ATAMM.

Systems implementing the ATAMM model consist of three components, the graph manager, the global memory, and a set of functional units or resources. The graph manager is responsible for ensuring that the overall system operates according to the ATAMM rules. The graph manager updates and monitors the status of the CMG. When a read transition of this graph is enabled, the graph manager assigns a functional unit from the queue of available functional units to perform the corresponding algorithm operation according to priority if more than one node is enabled. The graph manager updates the marking of the CMG using status information reported by the functional units. Therefore, the graph manager requires a communication path to each functional unit. The data corresponding to input and output signals for each AMG node are stored

in the global memory. Thus, the global memory also requires a communication path to each functional unit. The functional unit is the logical component that executes all three node marked graph (NMG) transitions of each algorithm operation. Therefore, the internal token marking at the "DP" edge is not important to the graph manager. The "PR" edge also provides only redundant information. The functional unit communicates with the graph manager to update the status of the CMG, and with the global memory to read and write data. The communications between the graph manager, the global memory, and functional units are asynchronous and are carried out by means of a communication channel. In order to ensure that all functional units have an identical copy of the graph data structure, a functional unit grabs the communication channel before changing the graph data structure. The updated graph data structure is transmitted to all functional units by a broadcast, and only then does the functional unit release the communication channel for other communication.

The graph manager and global memory may be distributed among all the functional units. This distribution of activities has the advantage of increasing the number of functional units in the system and at the same time improving the potential for achieving a higher degree of fault tolerance to processor failure. Also, a distributed global memory eliminates the need for shared memory among functional units.

The integration of the graph manager with the hardware's operating system constitutes the ATAMM Multicomputer Operating System (AMOS). The resource queue, global memory, and the algorithm marked graph provide the necessary support to AMOS. An AMOS controlled architecture consisting of IBM PC-AT's has been developed and tested to validate the ATAMM rules [13], [14]. In this testbed, a centralized graph manager and centralized global memory were utilized. Another testbed, called the Advanced Development Model (ADM) has also been developed [15]. The ADM system is composed of four functional units, utilizing a distributed graph manager and distributed global memory.

2.3 ADM system

A VHSIC ATAMM data-flow architecture, called the Advanced Development Model (ADM), has been developed [15]. The ADM system consists of four identical VHSIC 1750A processors which communicate over a dual PI-bus as shown in Figure 2.6. A 1553B communication module is also connected to the PI-bus. The 1553B serves as a gateway for input and output data flow from an IBM PC-AT. Communication over the PI-bus is accomplished by broadcasting and use of direct-memory access. The 1553B module is connected to the IBM PC-AT by a single line communication link (serial communication). Data are transferred between the 1553B module and the IBM PC-AT by synchronous communications. In addition to input and output, this link is used for fault injection, fault recovery, modification of the algorithm graph in real-time, and passing information back to the IBM PC-AT for testing purposes. The 1553B also acts as a source and a sink for the algorithm graph and thus is responsible for controlling the input injection rate to the 1750A processors and collecting the output. All processors, 1750A's and 1553B, communicate over an IEEE-488 bus to a Microvax computer which is used to download AMOS code, application programs, and files for debugging purposes. The performance of AMOS on the ADM has been characterized [16].

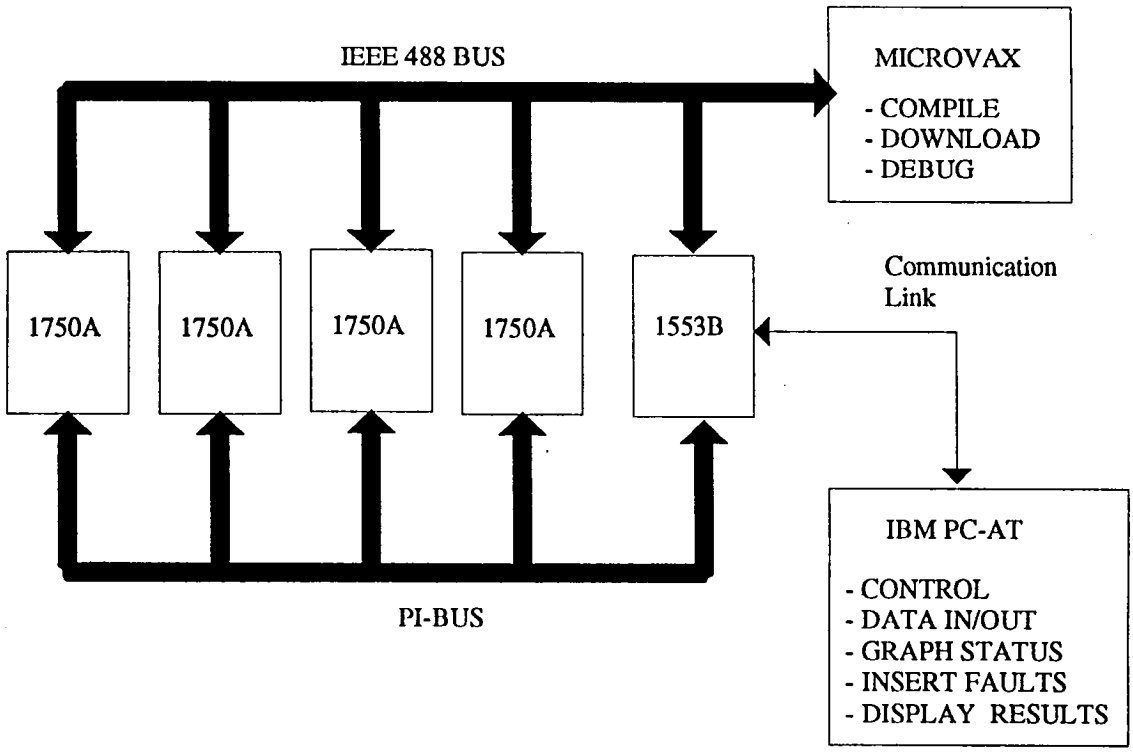

Figure 2.6. Layout of ADM System.

2.4 AMOS Description

An example of AMOS is the integration of the graph manager with the ADM hardware. The state diagram description of the AMOS is shown in Figure 2.7. AMOS is composed of five states: Idle, Examine, Execute, Test, and Update. Initially, all functional units awake in the state labeled Idle. A functional unit remains in this state until its identification number (ID) appears at the top of the resource queue (First- In-First-Out) of available functional units. Upon finding its ID, the functional unit undergoes a state transition to the Examine state. In this state, the functional unit actively monitors the status of the CMG until a read transition for an algorithm operation becomes enabled. Once an enabled read node is identified, the functional unit assigns itself to perform the algorithm operation. To progress to the next state, the Execute state, the functional unit grabs the PIbus, removes its ID from the top of the resource queue,

updates the CMG, reads the input data, and broadcasts the updated information to other functional units announcing that an algorithm operation has been initiated (fired). The functional unit then releases the PI-bus. This broadcast is called an "F" broadcast.

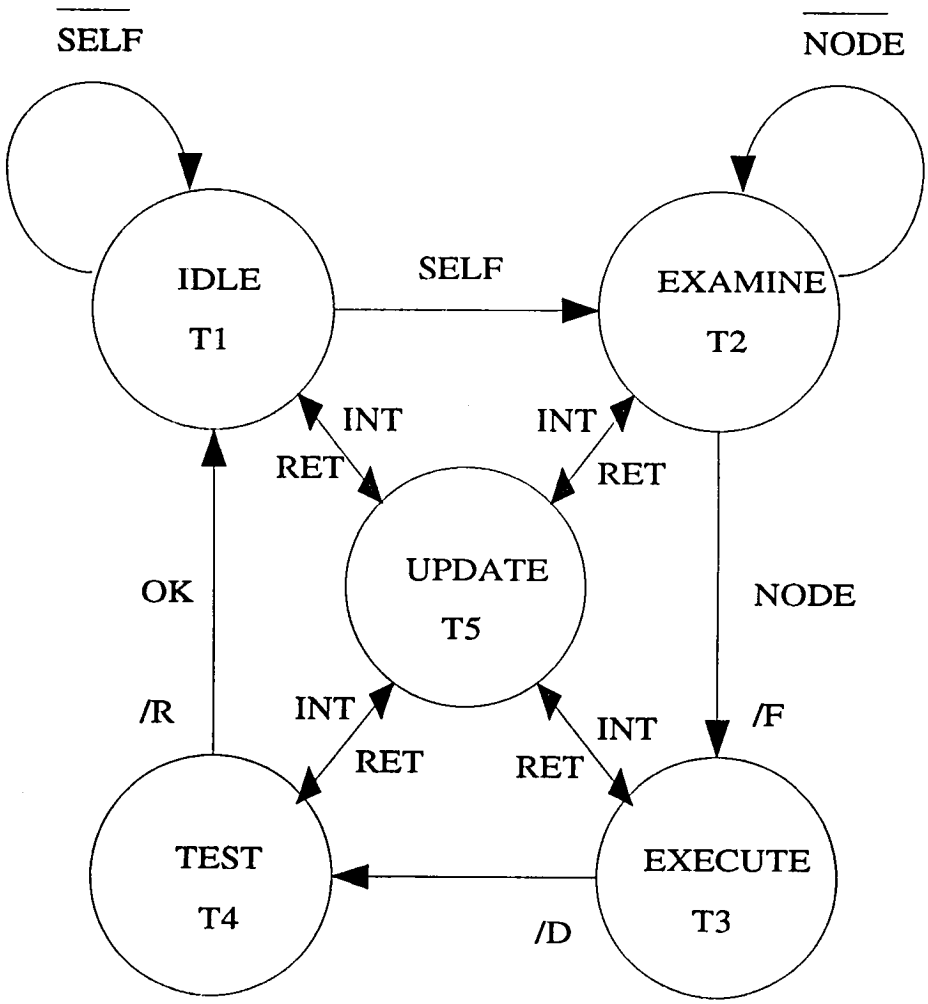

Figure 2.7. AMOS State Diagram.

The "F" broadcast contains the updated version of the CMG, the updated resource queue, and the ID of the functional unit processing the AMG node. This broadcast, as well as the two others discussed next, provide the status information necessary for the graph manager to maintain the status of the CMG. Since the graph manager is

distributed, this communication is especially important to ensure that all individual graph managers contain the same CMG marking.

The functional unit remains in the Execute state until the algorithm operation is complete. After completion of the algorithm operation, the functional unit undergoes another state transition to the Test state. It grabs the PI-bus, updates the CMG, writes the output data, and broadcasts the updated information to other functional units. This broadcast, termed an "D" broadcast, provides the updated CMG and the data generated by the primitive operation to the other functional units. The functional unit then releases the PI- bus.

The Test state corresponds to a diagnostic check of the functional unit. This state provides the means to remove a functional unit from the system for inspection during realtime operation. Upon a successful self-test, the functional unit places its ID at the bottom of the available queue and returns to the initial Idle state. This state transition is accompanied by a grabbing of the PI-bus, updating the resource queue, broadcasting the updated information, and a release of the PI-bus. This broadcast is named an "R" broadcast.

Since the operation of the system is asynchronous, the graph manager must generally be interrupt driven. While in any state, the CMG and resource queue in the global memory of a functional unit can be updated by "F", "D", or "R" broadcast from other functional units.

The "F", "D", and "R" broadcasts not only provide the communication necessary for integrity of overall system operation, but also the means to analyze the system performance. By labeling, time tagging, and storing information about each broadcast, such as the event (F, D, or R), the node number, and functional unit ID, the token movement within the CMG, and the token movement within the AMG, as well as functional unit activity can be reconstructed. Other measurements such as TBIO, TBO, and functional unit utilization and concurrency may also be extracted.

2.5 Fault Injection, Detection and Correction

Barry W. Johnson states that "The concept of fault tolerance has become increasingly important during the past decade because of the increased use of computers in the vital aspects of almost everyone's life. Computers are no longer confined to use as powerful calculators where their incorrect performance can produce little more than frustration and lost time. Instead, computers are now integrated into commercial and military aircraft flight control systems, industrial controllers, space applications, and banking systems. In each application, erroneous computer performance can be devastating to financial records, environmental safety, national security, and even human life" [17]. Two types of faults are modeled and handled in the ADM system. One is a self-test fault detected while in the Test state. Once a self-test fault is detected, the functional unit is assumed to remain in the Test state. The other fault is a data-error detected when in the triple modular redundancy (TMR) mode. The data-error fault is the result of a defective functional unit that generates an erroneous result.

To provide a degree of fault tolerance, a triple modular redundancy (TMR) scheme provides adequate redundancy in the system where a single error can be detected and corrected. The TMR approach implemented in the ADM system triplicates the processing and the data associated with each AMG nodes. An operation represented by a simplex AMG node is now represented by three AMG nodes with color extension to red, green, and blue. The rules for enabling and firing a simplex AMG node (Section 2.2) now applies to a TMR AMG node, where the three-colored AMG nodes are enabled and fired by three functional units, simultaneously. Each colored node triplicates its output data for each colored successor node. These data are also color referenced. In TMR mode, when a colored node fires, a majority vote is performed on all colored input data to select the correct data to process. The majority vote is also performed at the sink so that the correct data, as the final product of the graph, is chosen.

If only the detection of single errors is desired, a duplex implementation of the algorithm graph would suffice. However, in duplex mode, the AMG nodes are duplicated and are identified by the colors red and green. The rules for enabling and firing a duplex node is the same as its TMR counterpart. A graph may be executed in simplex, duplex, or TMR mode. A description of the AMG transformation from simplex to duplex and to TMR is shown in Figure 2.8.

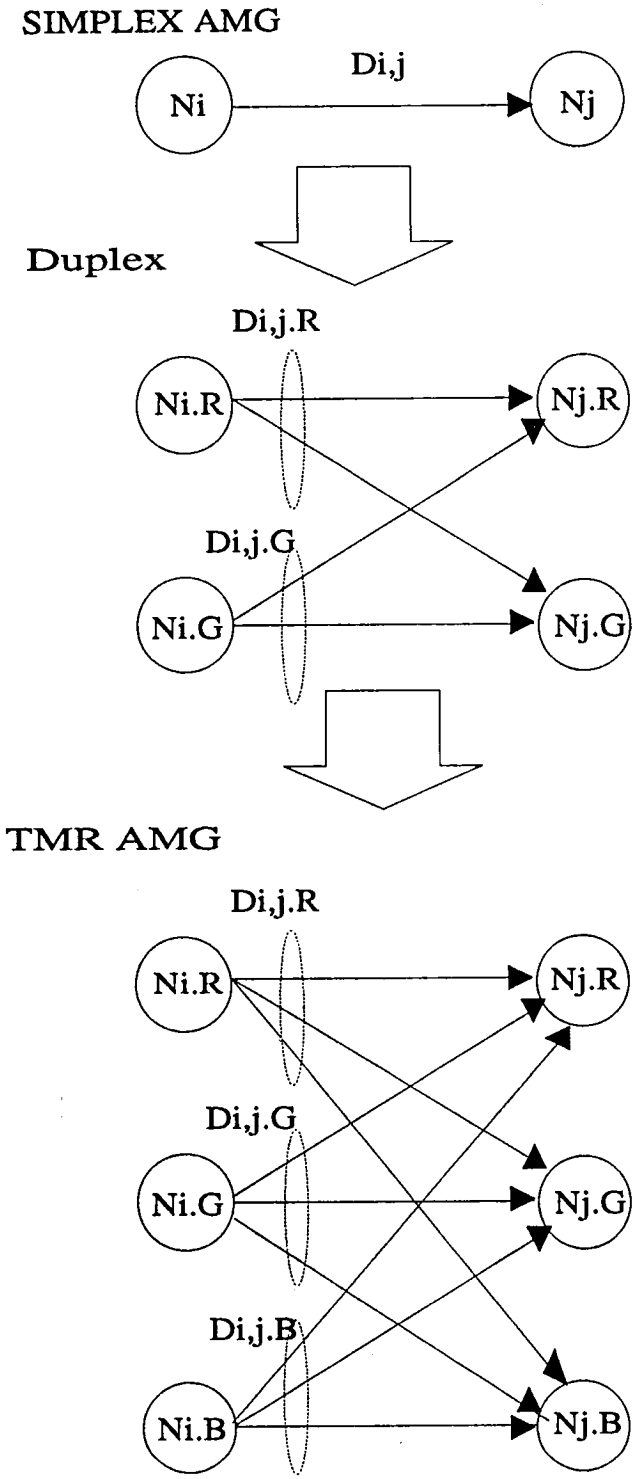

Figure 2.8. Simplex to TMR Transformation.

CHAPTER THREE

Simulator Development

3.1 Introduction

The development of the ATAMM Simulator is presented in this chapter. This Simulator allows the study of the behavior of algorithms in multicomputer data flow architectures operating in real-time based on the ATAMM model. The purpose of the Simulator is to permit an architecture-independent study of behavior, performance, and fault tolerance of a system without having to build a hardware prototype.

The Simulator represents a homogenous multicomputer data flow architecture. Object-oriented programming methodology, Section 3.2, lends itself to modeling different parts and relationship among the parts of a generic system. With this approach, simulation of the parallel execution of nodes of a graph by functional units is easily realized. The Simulator consists of six classes of objects. These objects are Graph, a set of nodes and edges; Graph-Manager, that represents the graph manager; Processors, which represents a set of functional units; Network, that represents a set of communication channels; PC, as the front-end of the ADM system; and Simulator-Kernel, that manages a multitasking environment for other objects to function. The Simulator- Kernel, Processors, Network, Graph, Graph-Manager, and PC are discussed in Sections 3.6, 3.7, 3.8, 3.9, 3.10, and 3.11, respectively. Section 3.5 is a discussion on the evolution of these objects and the relationship among them. The programming environment and language used in the development of the ATAMM Simulator are discussed in Section 3.3.

The Simulator's input and output requirement and formats are discussed in Section 3.12. As shown in Figure 3.1, the input to the Simulator is expressed as a

marked graph and the output of the Simulator is an FDT (Fire, Data, Time) file. The FDT file is a collection of time-tagged events which provide a means of evaluating the system performance and graph execution. Basic information in the FDT file includes the time occurrence of each event, name of the event, node number, node color, functional unit ID, and the current mode of the operation. The format of the FDT file is discussed in Section 3.8.4. The FDT file serves as the input to the Analyzer [18] which is a software tool that graphically displays algorithm and processor activities. The measurement capabilities of the Analyzer include graph latency, throughput, concurrency, resource utilization, and system overhead [18].

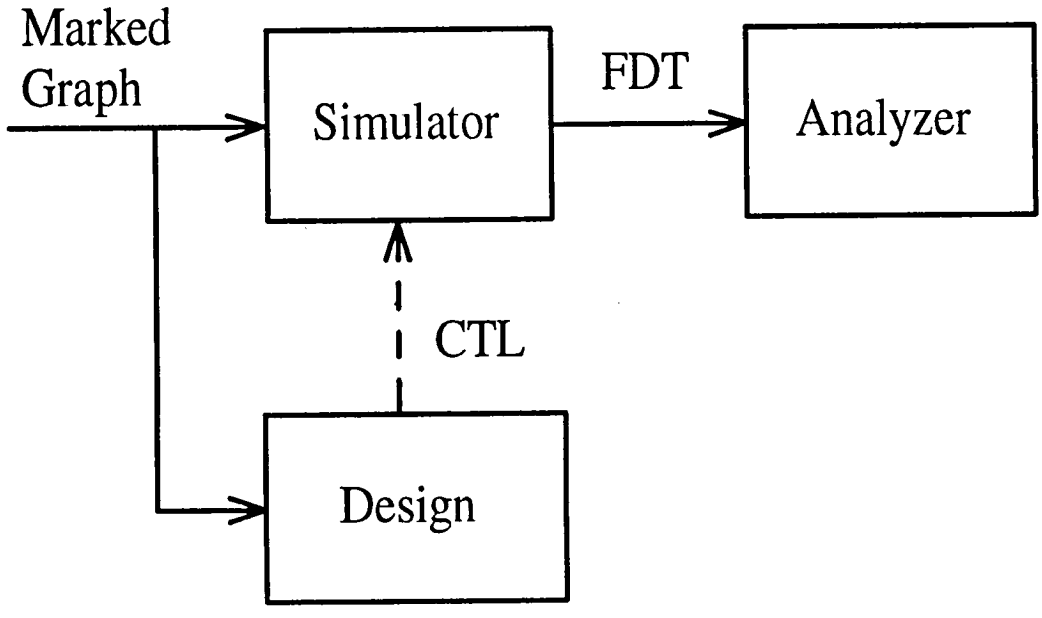

Figure 3.1. Flow of Information Among the Softwares Developed for ATAMM.

3.2 Object-oriented Programming

Structured programming flourished because it was efficient in terms of human resources. Building and testing programs in discrete pieces enabled large applications to be developed in less time with fewer bugs than their non- structured counterparts. In

addition, the run-time impact of structuring becomes less evident as a program grows in size. Object-oriented programming extends structured programming by encapsulating both data and their associated functions [19].

In traditional procedural languages like C or Pascal, the programmer defines data structures and writes functions and procedures to operate on the data. Although normally a correspondence exists between which functions operate on which types of data, most procedural languages offer no formal support for this correspondence; it is entirely the programmer's responsibility to manage such an abstraction.

In an object-oriented approach, *both data and operations that work with that data* are combined into a single logical unit known as an object. Dividing a program into objects encompassing both data and operations makes the program more closely represent the logical design that is being implemented. As a result, object-oriented programs are generally easier to understand and maintain than procedural programs.

Object-oriented programming is merely the art of breaking a program down and organizing it. In the case of structured programs, the primary concern is what the program is doing. A structured program is based on operations. When writing object-oriented programs, the program is organized around data types and their associated operations. It is a significant change in perspective; instead of functional hierarchies, there are data hierarchies. Programming in an object-oriented language involves creating objects and sending them commands or messages to do things.

Object-oriented programs are based on four concepts: classes, objects, methods, and inheritance. A "class" is similar to a Pascal RECORD. It describes an overall structure for any number of types based upon it. The main difference between a class and a record is that a class combines data fields (called "instance variables") and procedures and functions (called "methods") that act upon the data.

An "object" is a variable of a class. All objects derived from a class are considered members of that class and share similar characteristics of that class.

"Methods" are procedures and functions encapsulated in a class or object. Calling a method is referred to as "passing a message to an object." Object-oriented programs do most of their works by sending messages to objects.

Object-oriented programming lends itself to modeling different parts of a complex entity and the relationships among its parts. The objects can be defined and developed separately to ensure privacy of data, reusability, and readability. This also makes maintenance and debugging more manageable and systematic.

3.3 Programming Environment and Language

The implementation of the ATAMM Simulator requires a powerful software environment. The Simulator is developed in the Microsoft Windows[1] environment because of its object oriented programming capabilities including message passing, a non-preemptive operating system, and a vast library of graphics routines, especially the windowing capabilities. Using Microsoft Windows, the classes of objects are defined as separate windows (parent) and their subclasses as child windows. Every object, parent or child, can display all of the relevant information in its own independent window which allows the displays of different windows to be viewed at the same time. This provides an analysis capability that would otherwise be lost if it were only possible to view one display at a time.

The objects are defined and developed separately to ensure privacy of data, reusability, and readability. This also makes maintenance and debugging more manageable and systematic. The Simulator is written in the C programming language. The main reasons are: 1) it provides good data structures, control flow primitives, and a rich set of operators; 2) since C is a comparatively low level language having easy

---

1 Microsoft Windows is a trademark of Microsoft Corporation.

access to Processors-level information, it forms a good system programming language; and 3) Microsoft Windows library routines are generated in this language.

Other Microsoft Windows environment features include the capability to run more than one application in parallel, permitting the user to run more than one instance of the Simulator at the same time, thereby providing a means to simulate and compare two or more simulations simultaneously. As another example, the Simulator and the Analyzer programs can be running concurrently allowing an easier transition between them.

3.4 Animation

Traditional simulators require users to remember and type a great deal to specify the input/output requirements. This impedes learning and retention, especially by casual users. Utilizing the vast graphics library of the Microsoft Windows development kit, the Simulator was developed emphasizing recognition over recall: seeing and pointing over remembering and typing. Therefore, menus are extensively used instead of on-line commands. Most of the user-interaction is through dialogue boxes and mouse I/O in windows so only slight use of the keyboard is required. Also, to interact with the user during the simulation process, an animated display of the play of the graph, the movement of tokens, the status of functional units, and the status of the communication channel are provided.

3.5 Objects and Their Relationships

The main logical components or objects of the Simulator are, in part, a result of the ATAMM. Since the ATAMM is a set of rules by which an algorithm graph can be mapped to an architecture, the three main classes of objects: Graph- Manager, Graph, and Processors naturally result. Any system has some means of communication among its components; so the fourth object, Network, evolved. For ADM compatibility the fifth

object, PC, are added. In addition, there is a need to provide a management for arbitration among these objects, thus Simulator-Kernel are introduced. Interconnection among these entities is portrayed in Figure 3.2. Table 3.1 is a list and description of messages passed among these objects.

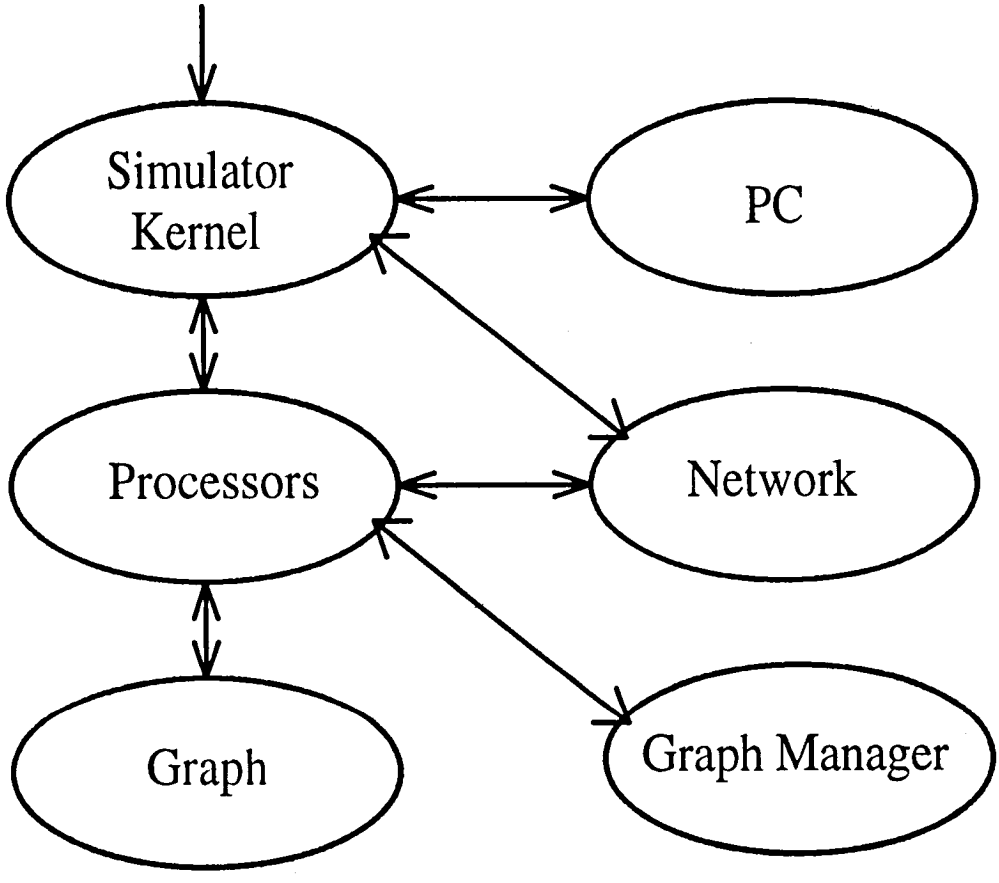

Figure 3.2. Relationships Among the Objects.

Message	Action
WM_STP	Load an STP file.
WM_GRAYMENU	Disable menu.
WM_SHOWMENU	Enable menu.
WM_NEWCHILDS	Create new children.
WM_WRITEMSG	Prompt a message.
WM_RUNONELOOP	Instruct all children to RUN.
WM_UPDATE	Update the graph structure.
WM_RUN	Perform the task.
WM_FIRE	Fire the node and issue an "F" message.
WM_DATA	An "F" broadcast.
WM_RESOURCE	An "R" broadcast.
WM_WRITING	An "D" broadcast.
WM_UPDATING	Updating the graph.
WM_BROADCAST	Broadcast the graph structure.
WM_RELEASE	Release the channel.
WM_CHANNEL	Channel is granted.
WM_GETCHANNEL	Try to grab the channel.
WM_REQUEST	Request a node to process.
WM_REREQUEST	Request for a node, with possible higher priority, to process.
WM_NODE1	An enabled AMG node found.
WM_NODE2	Highest priority enabled AMG node found.
WM_SOURCE_REQUEST	Request a source to fire.
WM_SOURCE_REREQUEST	Request for a source with higher priority to fire.
WM_SINK_REQUEST	Request a sink to fire.
WM_SINK_REREQUEST	Request for a sink with higher priority to fire.
WM_SHOW_TOKENS	Show current value of the tokens of the edges.
WM_WAITING	Wait until other FU's update their graphs.
WM_REMOVE	Remove the functional unit.
WM_RESTORE	Restore the functional unit.
WM_INSERT_EDGE	Insert a control edge.
WM_DELETE_EDGE	Delete the control edge.
WM_INCREASE_QUEUE	Increase queue size the edge.
WM_DECREASE_QUEUE	Decrease queue size the edge.
WM_1553_REQUEST	1553B requests for the channel.

Table 3.1. A list and description of messages passed among the objects.

3.6 Simulator-Kernel

The Simulator-Kernel provides, manages, and simulates the multitasking environment where the functional units can operate without conflict. Hence, this object is the operating system for the Simulator and thus the heart of this software. The arbitration among different objects is enforced in a non-preemptive manner, where every object is given enough time to accomplish its task. This is easily realized by employing object-oriented programming methodology.

The Simulator-Kernel object passes control to a constituent object and by doing so suspends itself. This gives the target object the full control over the system. Upon completion of its task, the target object returns control back to the Simulator-Kernel. Transfer of control is accomplished through the message passing capability of object-oriented programming. This process continues for all objects, in an orderly fashion, until simulation of the graph is complete.

The order in which the objects are invoked is as follows. First, the Processors object, described in the following section, is invoked. It, in turn, passes control to its constituents, functional units, in an orderly fashion. Second, the Network object is invoked to carry out its communication task. The Network, described in Section 3.8, in turn, passes control to its child objects. The Processors and the Network have the same behavior as the Simulator-Kernel toward their constituents. Finally, the PC, described in Section 3.11 is invoked to perform its task. The hierarchy of passing control to the lowest level objects, children objects, is portrayed in Figure 3.3.

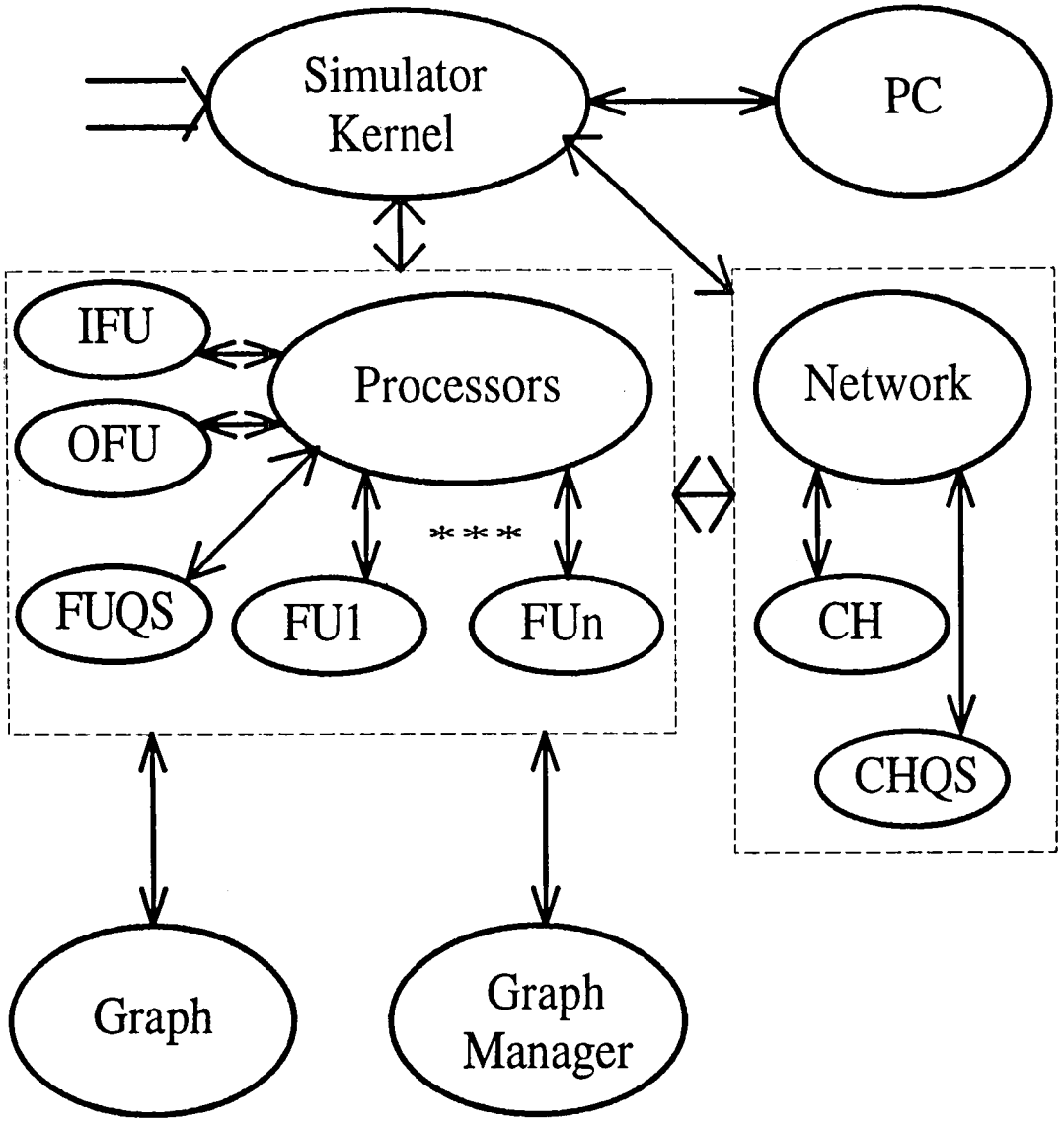

Figure 3.3. Hierarchy of Objects.

Thus far, the functionality of the Simulator-Kernel from an internal information viewpoint was described. However, another functional aspect of this object is its central role with respect to user interactions. The Simulator-Kernel object and all other objects that require user interaction have their own independent windows through which information may be passed and/or displayed. For these objects, the terms object and window are used interchangeably.

For user interactions, the Simulator-Kernel's window provides a set of menus. These menus are categorized according to their functionalities. The "File" menu allows the user to save and retrieve a set-up file (STP), discussed in Section 3.8.1, to open an output file to store the events of simulation of graphs for further study, and to specify the mode of operation. The mode of operation can be simplex, duplex, or triple modular redundancy (TMR). The "FileType" menu lets the user select the type of output file as either FDT or ADM. The speed of the simulation can be adjusted through a "Speed" menu to slow, medium, or fast. The duration of simulations of algorithm graphs can be specified by selecting the appropriate option from the "Run" menu. These options include the number of clock ticks, the number of events, and the number of data packets. This window as well as all other windows of the Simulator have two options in common. "Pause" is provided to pause the simulation process at any time. Selecting this option again will resume the simulation process. The other option common among the windows is the "Help" option where specific guidance for individual windows is provided.

Simulation of the graph may be triggered by specifying the number of clock ticks, the number of events reported, or the number of data packets fed to the graph. In any case, the Simulator keeps track of clock ticks, number of events, and number of data packets in and out of the graph. It also reports the current status of these activities for user's information upon receiving (acquiring) control of the system. Speed of simulation may be adjusted to fast, medium, or slow at any time. This provision is provided for animation purposes where the simulation of the graph is carried out at the desired pace.

Since this window is the heart of this software, existence of other windows depend on its existence, i.e., closing this window results in termination of the Simulator.

3.7 Processors

The Processors object treats its child objects in the same manner as its parent object, the Simulator-Kernel. The Processors is a set of three types of functional units.

These are functional units that operate on the source of the graph (IFU), functional units that operate on the sink of the graph (OFU), and up to twenty regular functional units (FU) that perform the tasks represented by the AMG node of the graph. The IFU and OFU are the corresponding computing elements of the ADM system, the 1553B. They are special functional units that do not operate on the AMG nodes of the graph. Because of ADM compatibility, the Processors is confined to one IFU and one OFU.

The Processors passes control to the functional units objects, in order, and by doing so suspends itself. The order in which these objects are invoked is now described. First, the IFU object is invoked to inject new data into the graph. The injection interval is determined by the sources of the graph. Second, the OFU object is invoked to fetch the graph output. Finally, the functional units, FU's, are called upon to carry out the execution of the AMG nodes of the graph.

Through Processors' window, the number of functional units can be specified to match a particular architecture. The number of functional units at the start of the simulation of algorithm graphs is the maximum number of resources for the duration of the simulation process. The number of available functional units of a system is crucial to the operation of the Simulator. More specifically, in the case of functional unit failures, reduction in the number of available resources may affect overall performance of the system and possibly change the mode of operation. The mode of operation is constantly monitored by the Graph-Manager object, Section 3.10. When the number of available functional units drops to two, the mode of operation is adjusted so that the highest mode possible is duplex. When there is only one active functional unit, the only possible mode is simplex. Nevertheless, recovery of functional units and thus increasing the number of available resources doesn't affect the mode of operation. The number of functional units, however, for practical reasons is limited to twenty [8]. Also, through the Processor's window, the time it takes a functional unit to conduct a self-test may be adjusted. Since

the Processors object represents a homogenous system, the test time is identical for all functional units.

To inform the user of the status of the system, contents of the available queue (QUEUE), the working pool (WORK), the diagnostics pool (DIAG), and the recovery pool (RECOV) of functional units are displayed at run time. The colors used to distinguish QUEUE, WORK, DIAG, and RECOV are green, red, yellow, and white, respectively. This status reporting is accomplished by another child object of the Processors' called functional units queues (FUQS).

3.7.1 Functional Units (FU's)

To carry out execution of an AMG node of any kind, upon availability, a functional unit, of any type, has to communicate with the graph manager to find an enabled node. To fire the AMG node, the functional unit has to grab the channel in order to read the input to the node and to broadcast the updated graph, an "F" broadcast. To grab the channel all functional units must compete. The channel is granted based on the specified protocol of the defined architecture. Section 3.8 is a complete description of communication network protocols. The hierarchy of flow of messages among functional units, the Graph, and the Graph- Manager objects is portrayed in Figure 3.4.

To complete execution of an AMG node, the attached functional unit goes through a sequence of states as depicted in Figure 3.5 [20]. These states define the operating system characteristics of the ATAMM Multicomputer Operating System (AMOS). The state diagram of functional units is described in the following Section.

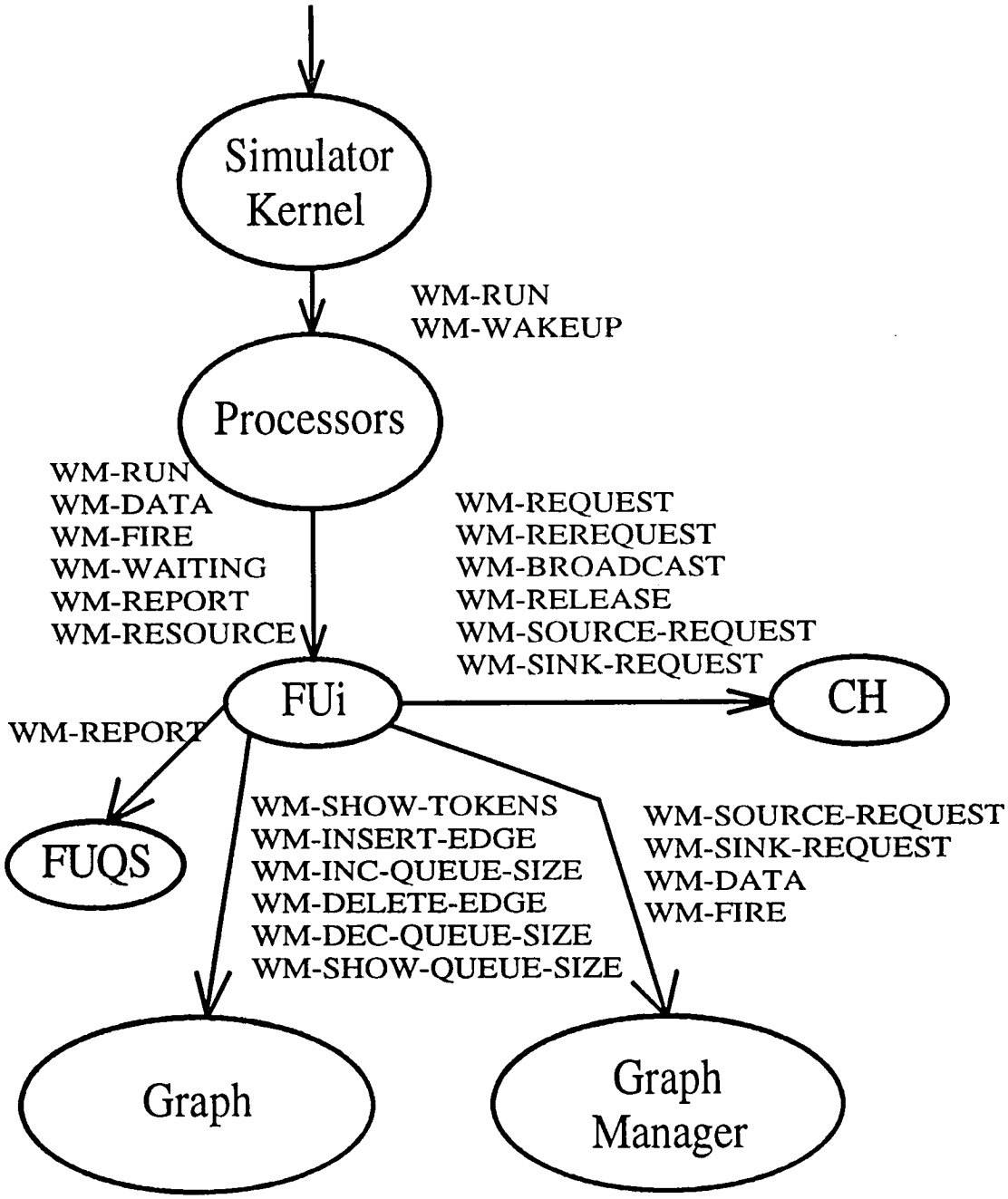

Figure 3.4. Hierarchy of Flow of Messages Through Processors Objects.

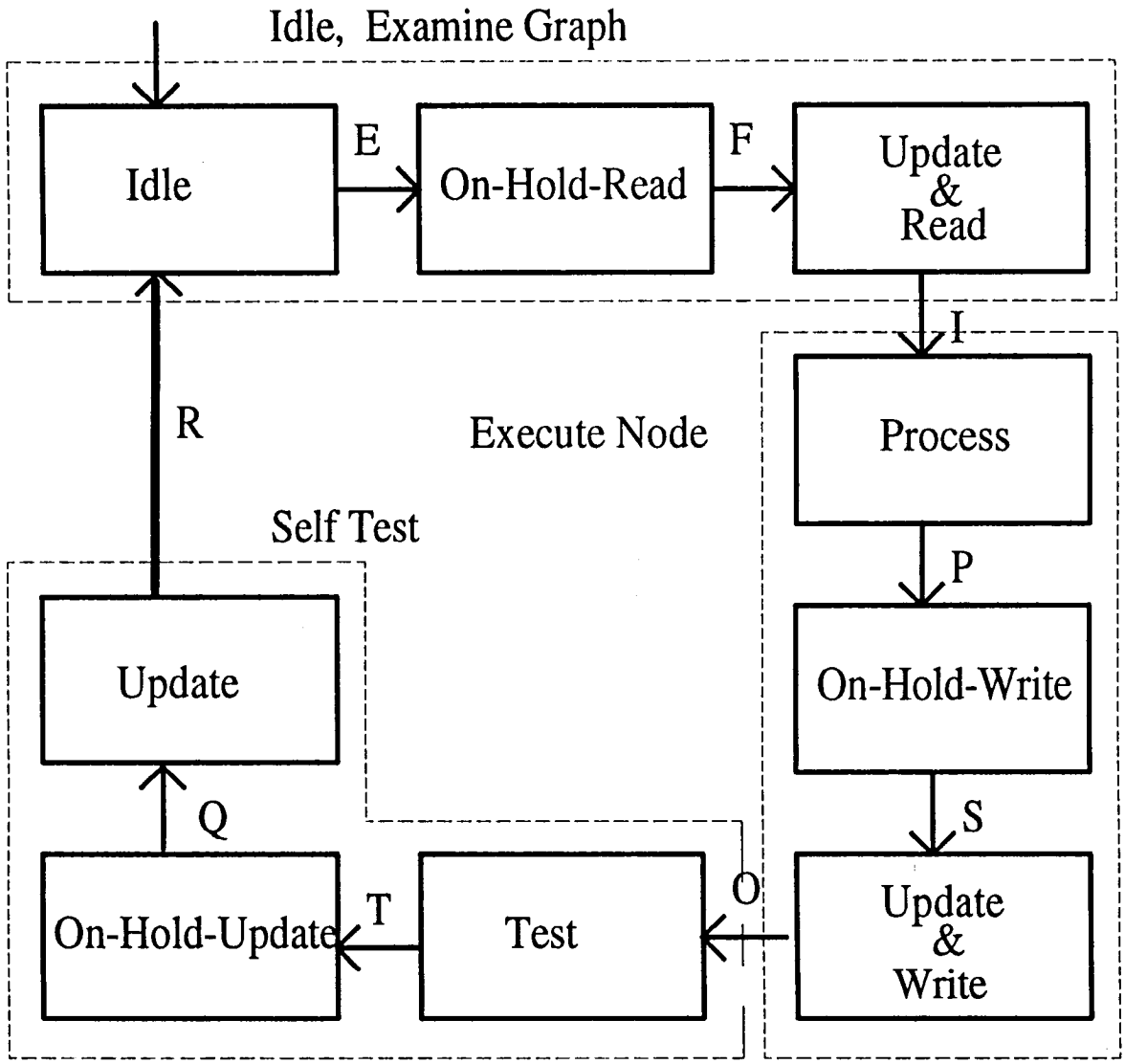

Figure 3.5. State of a Functional Unit.

The current state of a functional unit, labeled and time tagged, along with other information about the current state, such as the node number, and functional unit ID are stored in an FDT file upon entering the state. This information can later be analyzed by reconstructing the token movement within the CMG and the token movement within the AMG, as well as functional unit activities by the Analyzer [18]. These operating system states are also displayed in the window of each functional unit. Every functional unit

reports its current status by coloring the appropriate rectangle representing the current state red. If a functional unit is disabled, e.g., in the case of a self-test failure, its state cannot be determined.

3.7.2 FU State Diagram Description

Idle

When idle, the functional unit continuously examines the queue of available functional units (QUEUE), first to check if there are at least as many functional units in the QUEUE as the MODE of the system and second, to check if it is one of them. When it finds itself at the top of the QUEUE, it searches for an enabled AMG node based on priorities assigned to the nodes. An enabled node is detected by examining all input and output edges of the node. This search continues until an enabled node is found. Having an AMG node to execute, the functional unit selects a colored-node, based on its position in the QUEUE, to fire.

On Hold Read (grab channel)

To read inputs associated with the node, the functional unit has to get hold of a channel. The duration of this state depends on the traffic and communication channel protocol.

Update and Read

After establishing a communication link, the functional unit conducts a second search for enablement of nodes with higher priorities than the previously enabled nodes. Selecting a node with the highest priority, the functional unit migrates from the QUEUE to the pool of working functional units (WORK). It then broadcasts the updated graph. This broadcast is called the "F" broadcast. After reading the node's input data, the functional unit releases the communication channel. After reading every input, depending on the MODE of operation, the functional unit may have to vote and select the

proper input. Specifically, in TMR mode, the functional unit votes on the three sets of inputs and chooses the correct set for processing.

In the ADM system, since the inputs to the nodes are stored in the local memory of the functional unit, the functional unit doesn't need to hold on to the communication channel. The functional unit, therefore, releases the channel before reading the input data of the node.

Process

In this state, the functional unit executes the application program. To do so, control is passed to the application program. Upon completion of the task, control is passed back to AMOS. The duration of this state is the same as the execution time of the application program.

On Hold Write (grab channel)

To write the generated outputs, the functional unit has to get the hold of a channel. The duration of this state depends on the traffic and communication channel protocol.

Update and Write

After establishing a communication link, the functional unit migrates from the WORK queue to the diagnostics queue (DIAG). In this state the functional unit writes the output data to the proper locations. It then broadcasts the updated graph. This broadcast is termed the "D" broadcast. If an error were detected at the Read state, the color of the node and ID of the functional unit responsible for the error are broadcast. The communication channel is then released.

Test

In this state the functional unit performs a self-test. Upon completion, the functional unit requests for a channel. Duration of this state depends on the test routine.

On_Hold_Update (grab channel)

To let the system know about its availability to undertake another task, the functional unit needs to grab a channel.

Update

After establishing a communication link, the functional unit migrates from the diagnostics queue to the queue of available functional units, if the self-test were successful. Otherwise, it removes itself from the diagnostics queue. It then broadcasts the updated graph. This broadcast is called the "R" broadcast and releases the communication channel.

The IFU and OFU are special functional units, and therefore only go through some of the operating system states to accomplish their duties. Specifically, the IFU goes through the Idle, On_Hold_Write, and Update and Write states, while the OFU goes through the Idle, On_Hold_Read, and Update and Read states. The state diagrams of the IFU and OFU are shown in Figures 3.6 and Figure 3.7, respectively.

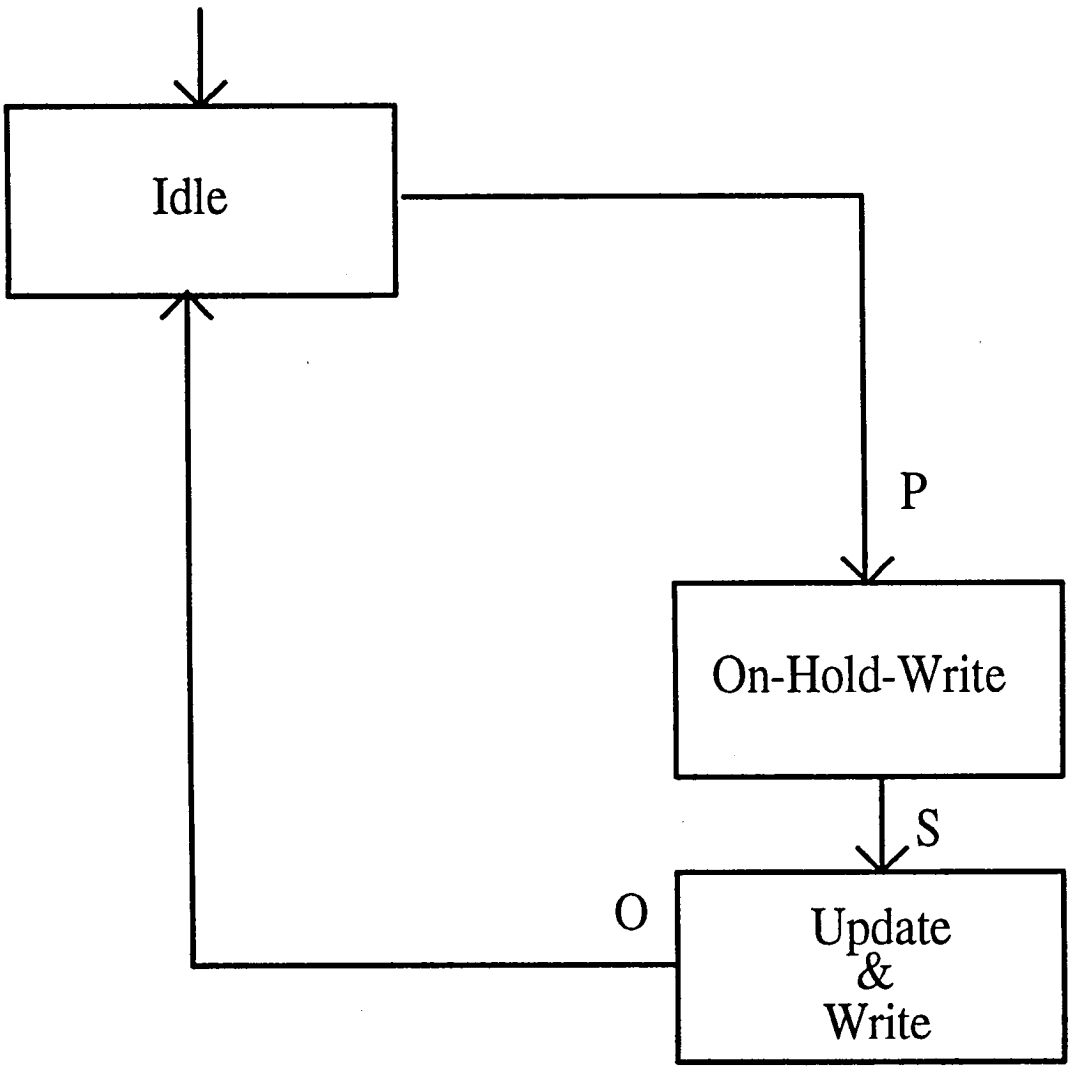

Figure 3.6. States of an IFU.

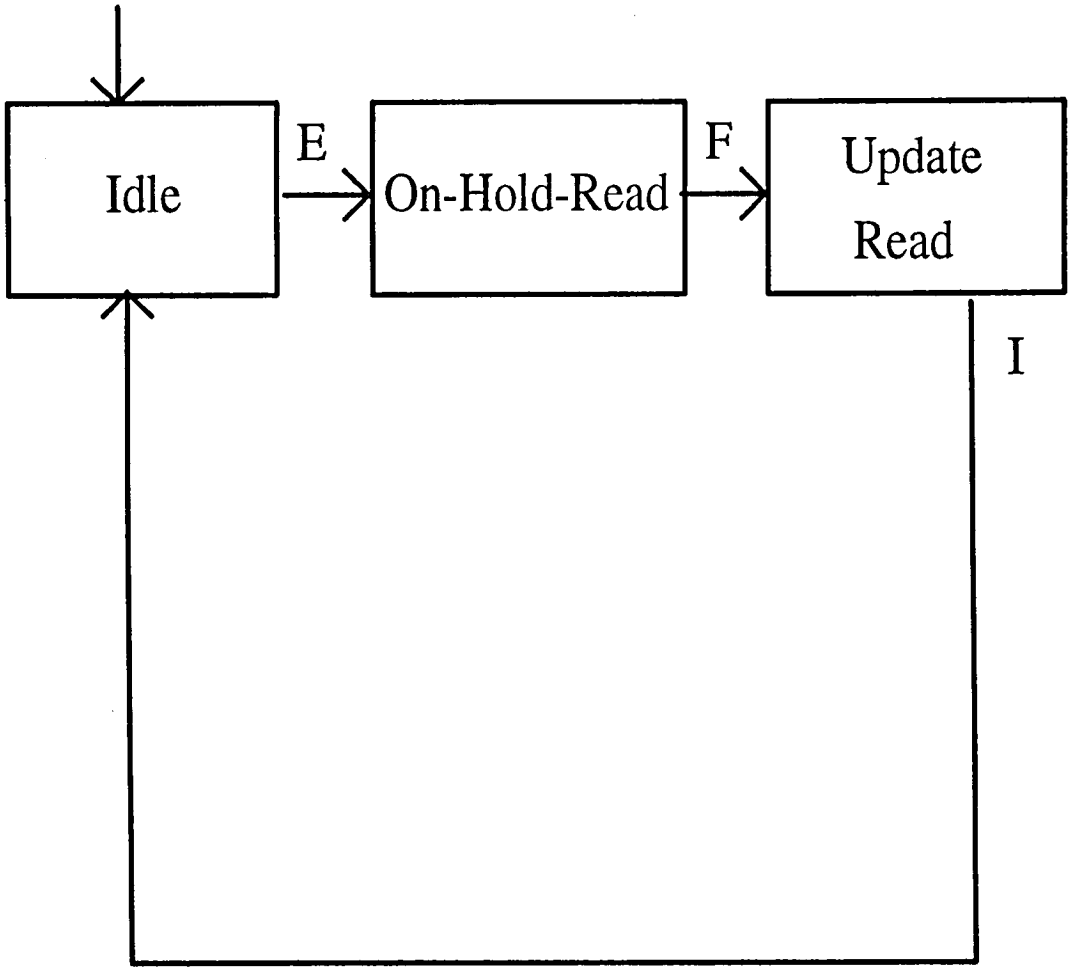

Figure 3.7. States of an OFU.

3.8 Network

The Network object treats its child objects in the same manner as its parent object, the Simulator-Kernel. Due to ADM compatibility, the Network is confined to one communication channel corresponding to the PI bus of the ADM system. The Network passes control to the channel and by doing so suspends itself. To grab the channel all functional units must compete. The hierarchy of message flow among the Network, the Channel, and functional units objects are portrayed in Figure 3.8. The channel is granted based on the specified protocol of the defined architecture. These

protocols include priority and first-come, first-serve strategies. In the priority protocol, the physical proximity of the functional units is the criterion for granting the channel. The priority of the functional units is identified by their unique ID's. This protocol is adopted by the ADM system. Since the IFU and OFU are closest to the PI bus semaphore, they have the highest priority. In the first-come, first-serve or first-in, first-out (FIFO) protocol, the channel is granted to the functional units based on their request time. The channel reports its current status, Idle or Busy, by coloring red the appropriate rectangle representing the current state red. The state diagram of the communication channel is shown in Figure 3.9.

To accomplish its task, the Network stores all requests for the channel and then based on the criteria imposed by the specified communication protocol grants a request. These requests are stored in the request-queue of the Network. To inform the user of the status of the Network, the content of the request-queue is displayed at run time. This is accomplished by another child object of the Network called channel queues (CHQS).

Through Network's window, such parameters as GrabTime, UpdateTime, BroadcastTime, and WaitTime may be adjusted to reflect the desired network's characteristics. For a definition of these variables refer to the help files provided with the Simulator.

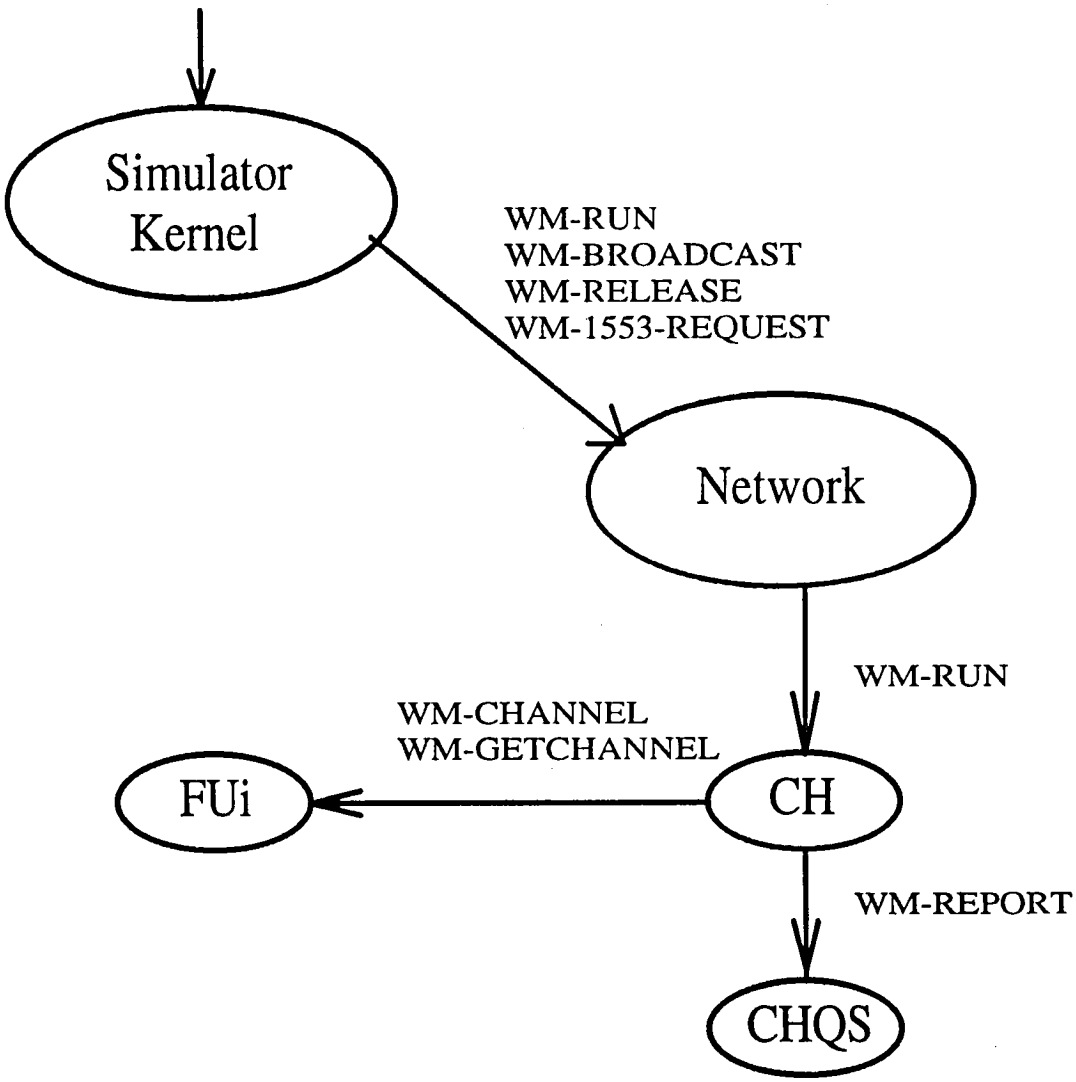

Figure 3.8. Hierarchy of Flow of Messages Through Network Object.

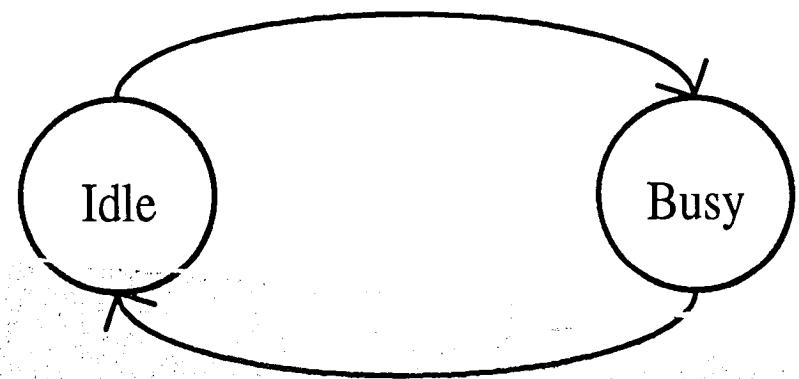

Figure 3.9. Communication Channel State Diagram.

3.9 Graph

An algorithm marked graph is the essential and necessary part of the ATAMM model and the Simulator. An algorithm marked graph consists of a set of nodes, a set of edges, a set of sources, and a set of sinks. The Graph object provides the means to create and update algorithm graphs, with the desired characteristics, for simulation purposes. The ATAMM Simulator, as a research and design tool, is flexible and powerful enough to simulate a graph at any marking state. The Graph object provides the necessary means to initialize both data and control edges with the tokens so that the behavior of the algorithm graphs starting at certain marking states can be studied. While simulating a graph, the Graph object displays movement of tokens on the data and control edges. The animated display of the marking of graphs provides a means to symbolically observe the play of the algorithm graph on the specified system. Since the Simulator is capable of simulating multiple independent graphs where each graph may have multiple sources and multiple sinks, the Graph object is designed accordingly so that multiple graphs may be created. Regardless of the number of graphs, the total number of nodes is confined to at most twenty. As stated in chapter 2, this restriction is due to practical reasons. The number of sources and sinks are at most ten, and number of edges at most sixty. Since the Simulator is independent of grain size, these numbers are arbitrary.

Although the final product of this object, algorithm graphs, are accessible by other objects, other objects cannot modify the algorithm graphs. Any run-time modification to the graphs, e.g., adding a control-edge, is performed by the Graph object upon receiving the corresponding messages with the necessary parameters. A number of messages passed to the Graph object are listed in Figure 3.4 and Table 3.1.

To create a graph, the Graph object's window provides a set of menus for user interactions. These menus are categorized according to their functionalities. The "File" menu allows the user to save and retrieve a graph file (GPH). The GPH file is discussed

in Section 3.12.1. To draw an element of the graph: source, node, sink, edge, or control-edge, the corresponding item may be selected from the "Draw" menu. The "Edit" menu, however, lets the user select, delete the selected element of the graph, and clear the entire drawing. The "Update" menu provides the means to specify the timing parameters of the nodes, sources, and sinks, the number of tokens on any edge, and the queue size of any edge. The timing parameters include read-time, process-time, and write- time for the nodes, write-time and injection-time for sources, and read-time for sinks.

Although the primary color used in drawing a graph is black, green is used to identify the control-edges and red is used to indicate the selected element of the graph that may be deleted.

To draw a graph, having selected the desired element from the "Draw" menu, clicking the left-mouse button results in creation and display of that element of the graph centered at that point. Nodes are represented by circles, sources and sinks by rectangles with their names in them, and edges by segmented lines. The queue of an edge is displayed as a square centered on the last segment of that edge with the initial size of one displayed above the square. Also the initial token of that edge, zero, is displayed inside of the square representing the queue. Each element of the graph has its unique ID. When displaying a source, a node, or a sink, their ID is also displayed within their representative shapes. Successive clicking of the left-mouse button results in creation and display of that element with its consecutive ID number. The ID's of all elements start with one.

The graphs created in this module are suitable to run in any mode. The data structures of the graph are listed in Figure 3.10. These data structures are essentially the same as the data structures used in the development of AMOS. However, some additional parameters such as "location" and "TERM_KIND" are added for drawing and simulation use. The AMOS data structure is described in Appendix A.

```
typedef struct {
        short           ID,
                        NEXT,
                        enable_ctr,
                        busy_ctr,
                        done_ctr,
                        id[3],
                        input_summary[3],
                        output_summary[3] ;
        long            read_time,
                        process_time,
                        write_time,
                        test_time ;
        short           inputs[3],
                        outputs[3] ;
        RECT            location ;
        } nodes_rec ;

typedef struct {
        short           ID,
                        NEXT,
                        KIND,
                        token,
                        segment,
                        edge_color,
                        items,
                        output_width,
                        next_input,
                        next_output,
                        terminal,
                        initial,
                        TERM_KIND,
                        INIT_KIND,
                        LINE_SEGMENTS ;
        POINTPTS [ MAX_SEGMENT ] ;
        RECT            location [ MAX_SEGMENT - 1 ] ;
        RECT            q_location ;
        } edges_rec ;
```

Figure 3.10. Data structures of the Graph object.

```
typedef struct {
        short           ID,
                        NEXT,
                        enable_ctr,
                        busy_ctr,
                        done_ctr,
                        output_summary[3] ;
        long            write_time,
                        process_time,
                        inject_time ;
        short           outputs [3] ;
        RECT            location ;
        } sources_rec ;

typedef struct {
        short           ID,
                        NEXT,
                        enable_ctr,
                        busy_ctr,
                        done_ctr,
                        input_summary[3] ;
        long            read_time;
        short           inputs [3] ;
        RECT            location ;
        } sinks_rec ;
```

Figure 3.10. (continued) Data structures of the Graph object.

3.10 Graph-Manager

The graph manager is responsible for ensuring that the overall system operates according to the ATAMM rules. The Graph-Manager object, representing the graph manager of the ATAMM, updates and monitors the status of the CMG. When a read transition of the algorithm graph is enabled, the Graph- Manager assigns a functional unit from the queue of available functional units to perform the corresponding algorithm operation according to priority if more than one node is enabled. The Graph-Manager updates the marking of the CMG using status information reported by the functional units. Status information is reported to the Graph-Manager via passing messages. A number of these messages are depicted in Figure 3.4 and Table 3.1. The mode of the operation is constantly monitored by the Graph-Manager object. In the advent of functional unit failures, when the number of available functional units drops to two, the mode of operation is adjusted accordingly so that the highest mode possible is duplex. When there is only one active functional unit, the only possible mode is simplex. Nevertheless, the recovery of functional units and thus increasing the number of available resources doesn't affect the mode of operation.

The graph manager constantly monitors the well-being and number of the available functional units. When the number of active functional units changes, the graph manager identifies a new operating point corresponding to the current number of available functional units. System operation at the new operating point is achieved by adjusting the injection time of the sources and by modification of the AMG through the addition or deletion of the control edges and increasing or decreasing the queue size of the edges. To accomplish this task, a set of actual operating points are selected by identifying as many operating points as the number of available resources. Each such point specifies the system time performance, TBIO and TBO, for a particular number of available resources. The set of actual operating points selected in this way is compiled in a control file (CTL), discussed in Section 3.12.3.

The graph manager may either be centralized or distributed, as stated in Chapter 2. The graph manager, in the ADM system, is distributed among the functional units, 1750A's. In this simulation software, the graph manager, represented by the Graph-Manager object, is logically partitioned from the computing elements, and therefore, can be thought of either as part of the functional units, a distributed graph manager, or as a centralized entity.

Since the Graph-Manager doesn't require user interactions, it doesn't have a window and is incorporated in the Graph window.

3.11 PC

The PC object emerged due to the need for ADM system compatibility. The hierarchy of the flow of messages among Simulator-Kernel and the PC objects is portrayed in Figure 3.11. The PC object represents the front-end of the ADM system. This object supplies the algorithm graph with inputs and stores the outputs of the graph in an output file. The inputs are periodically supplied to the graph at the injection intervals specified by the sources of the algorithm graph. The outputs, however, are periodically stored at communication intervals, where the communication interval is assumed to be shorter than the injection interval.

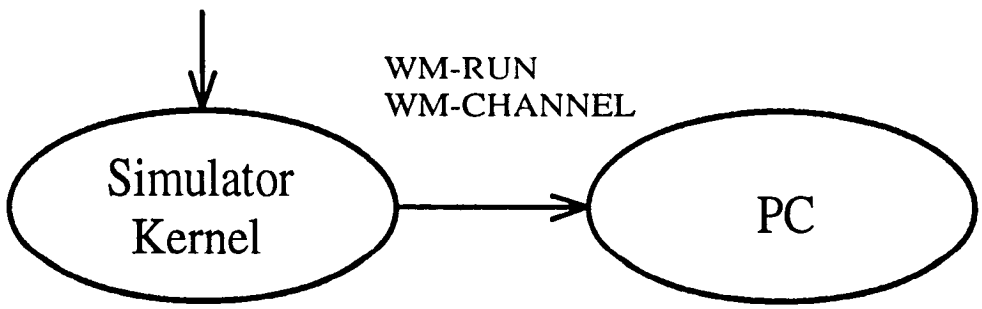

Figure 3.11. Hierarchy of Flow of Messages Through PC Object.

The PC object also is a means to control performance of the graph based on the current number of active resources. To control performance of the graph, the status of the functional units is constantly monitored. A change in the number of active resources invokes a new performance operating point and the graph is modified appropriately by adding or deleting control edges and/or increasing or decreasing queue sizes of the edges. The appropriate actions are invoked from a control file (CTL), Section 3.12.3. This file, however, must be first loaded prior to starting simulation of the graph.

Injection of faults to the system is conducted via the PC object and through the control-blocks. The self-test fault is simulated by directing a functional unit to fail during the self-test. However, the failed functional unit is assumed to be able to detect its own failure. As the result of this fault, the functional unit will not advance to the next state. This, in effect, will result in removal of the functional unit from the system. Since the maximum number of resources during the execution of algorithm graphs is a fixed number, adding a functional unit to the system is possible only after removal of a defective functional unit. Adding a functional unit to the system corresponds to the replacement of a defective functional unit. To add a functional unit to the system, the corresponding functional unit is directed to bypass the Test state and thus join the working force of the system. The functional unit then accomplishes this task by inserting itself in the queue of available resources, an "R" broadcast. The particular action taken by a functional unit is conveyed by the CTL file.

The off-line user interface to the system may be turned on by choosing the FAULT option of the PC's menu. When in off-line control mode, the communication interval may be adjusted to meet the desired need.

3.12 Inputs and Outputs

The primary input to the Simulator is a marked graph, as illustrated in Figure 3.1. The marked graph may either be developed through user interaction or loaded from an

existing file via the Graph window. The Simulator, however, doesn't necessarily need additional input files. It can run a graph (or multiple graphs), with user interactions and without loading any files. Nevertheless, there are management files that can be loaded to speed up user interactions. These files are identified by their extensions and are depicted in Figure 3.12. These files contain information, as designed by the user, about systems (STP), graphs (GPH), and injection control, control blocks, performance plane characteristics, and fault tolerance schemes (CTL). These files and their formats are discussed in the following sections.

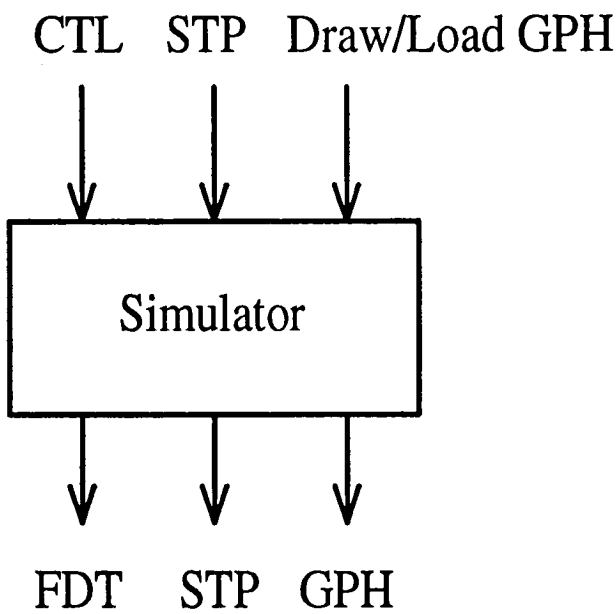

Figure 3.12. Simulator Inputs and Outputs.

The Simulator has two means of output. The screen is the primary output of the Simulator where an animated play of a graph is displayed, in real-time, via the Graph's window. The animated activities of the system are portrayed via the functional units' and communication's windows. The other means of output is a file, either in the FDT or

ADM format, where the time-tagged status events generated by the functional units are stored for further analysis by the Analyzer [18].

3.12.1 GPH File

The graph (GPH) files contain a list of information pertaining to the graphs developed via the Graph object's window. Basic information conveyed in the GPH file includes number of nodes, edges, sources, and sinks, their logical location in the Graph object's window, and relationships among them. The information about the edges reveal the ID and the type of entities that the edges link. Also, other parameters associated with each edge are a color (for TMR use), type of edge, number of line segments that constitute the edge, size of the queue of the edge, and number of tokens on that edge. The information about the nodes, the sources, and the sinks include such timing parameters as read, write, and process time. The graphs are created and stored in the TMR form. There is a one-to-one correspondence between the graph structure in memory and the information stored in the GPH file. This makes the loading of the graph into memory faster. Every record of the GPH file consists of a descriptive name that identifies a parameter followed by the value of that parameter. The format of the GPH file and its records are described in the BNF[2] notation and are presented in Figure 3.13. Figure A.1 is an example of an GPH file.

[2]Backus-Naur's Formula (BNF).

Format: **{<separator> <CR> {<string> <integer> <CR>}^{n}}^{p}**

where n is an integer and represents number of nodes, sources, sinks, or edges of the graph for a particular p, and p is an integer with a value of four. The corresponding value of n for a particular value of p is defined below.

p	n
1	number of nodes
2	number of edges
3	number of sources
4	number of sinks

separator = a string of length twenty. This string is used to separate different entities of the graph.

string = a descriptive name of up to ten characters.

integer = an integer value corresponding to the string identifier.

Figure 3.13. GPH file format using the BNF notations.

3.12.2 STP File

The set up (STP) file contains the key characteristics of a system. Some of this information addresses the mode of operation, while other parts reflect some of the characteristics of the Processors' and the Network's objects. The parameters related to the Processors' object include the number of functional units and the test time of the functional units. The parameters related to the Network's object consist of grab time, broadcast time, update time, wait time, and type of the communication protocol. The format and an example of an STP file are presented in Figure 3.14 and Figure A.2, respectively.

Format: {<string> <integer> <CR>}^{9}

string = an identifier of up to ten characters.
integer = an integer value corresponding to the string identifier.

Identifier	Value
MODE	Mode of operation 1 -> Simplex 2 -> Duplex 3 -> TMR
NumofFUS	Number of functional units, 1 to 20.
Protocol	Type of protocol employed by the Network 0 -> FIFO 1 -> Priority
Grab_Time	Minimum amount of time it takes to grab the channel semaphore.
BDCT_Time	Minimum amount of time it takes to broadcast the updated graph.
Test_Time	The time it takes to complete a self test by a functional unit.
Updt_Time	The time it takes to update the graph.
Wait_Time	The time it takes to service an interrupt.

Figure 3.14. STP file format.

3.12.3 CTL File

The control-blocks contain the necessary information to improve the performance of an algorithm graph under limited availability of resources. The control-blocks of an algorithm graph are stored in a control-blocks (CTL) file. The CTL file contains all the necessary information about injection of inputs and faults and management of the graph via control- edges and queue sizes of the edges.

This file contains three types of information, an injection table (T_TABLE) of up to twenty entries, one entry per functional unit, a fault table (FAULT_TABLE), and a control table (CTRL_TABLE) of up to twenty elements, one element per functional unit. This information is collectively termed "control-blocks". The information conveyed by the FAULT_TABLE includes functional unit ID, initial and terminal nodes of an edge, and the type of actions to be taken. The actions include inserting or deleting of control-edges between the initial and terminal nodes, increasing or decreasing the queue size of an edge, identified by the initial and terminal nodes, fault injection, and recovery of the functional units.

This file must be loaded in prior to starting the simulation process if changes in the number of resources are expected. The format and an example of an CTL file are presented in Figure 3.15 and Figure A.3, respectively.

Format:	**{<I_T>^{20}} <CR>}**	Injection Table.
	{<D_P F1 F2 F3> <CR>}^{10}	Fault Table.
	{<N> <CR>}	
	{{<Action Initial Terminal Size> <CR>}^{13}}^{N}	Control Table.

Where I_T, D_P, F1, F2, F3, N, Action, Initial, Terminal, and Size are integers.

Identifier	Description
I_T	Injection rate of new operating point.
D_P	Data Packets fed into the graph.
F1	ID of FU to be killed during self-test.
F2	ID of FU to be added to the system.
F3	ID of FU to be removed from the system.
N	Number of control blocks. This number must be same as maximum number of FU's at the start up.
Action	Type of modification to the graph 0 -> Stop. 2 -> Insert a control edge. 3 -> Delete the control edge. 4 -> Increase queue size of the edge. 5 -> Decrease queue size of the edge.
Initial	ID of Initial node.
Terminal	ID of the terminal node.
Size	Desired size of the queue of the edge specified by Initial and Terminal.

Figure 3.15. CTL file format.

3.12.4 FDT File

The FDT file is designed around the Analyzer and is described in detail in [18]. Nevertheless, for convenience the description and format of the FDT files is restated here.

Evaluating the performance of a concurrent processing system based on the ATAMM requires information concerning the state of each processor and the algorithm with respect to time. The broadcast of events as a processor progresses through the states of AMOS was discussed in Section 2.7.2. By knowing the graph structure, these events imply information about the movement of tokens within the CMG. Therefore, by recording these events along with the time of occurrence, processor and algorithm activity can be reconstructed.

The FDT (Fire, Data, Time) file contains a list of information pertaining to each AMOS broadcast, in order of occurrence, which provides a means of evaluating the system performance and graph execution. Basic information in the FDT file includes the time occurrence of the event, name of the event, block number, node color, FU ID, and the current mode (simplex, duplex, TMR) of the system.

The capability of evaluating overhead is made possible by adding information to each AMOS broadcast. This information is the time spent waiting for a communication channel and the time spent updating the graph structure for the broadcast. The update time also includes the read and write time associated with processing a node when attached to the respective "F" and "D" broadcasts. The format of the FDT file is presented in Figure 3.16.

Format: **{<T,Time,M,Mode,Event,N,Node,C,Color,Resource> <CR>}^{P}**

Where T, M, N, C, P, and Event are characters and variables Time, Mode, Node, Color, and Resource have integer values.

Identifier		Description
Time		Time of the event
Mode		1 -> Simplex
		2 -> Duplex
		3 -> TMR
Event		Name of the event
Node		AMG node number
	E	ON_HOLD_READING; channel wait for "F" broadcast
	F	READING; update for "F" broadcast
	I	PROCESSING; "F" broadcast
	P	ON_HOLD_WRITING; channel wait for "D" broadcast
	S	WRITING; update for "D" broadcast
	O	TESTING; "D" broadcast
	T	ON_HOLD_RETURNING; channel wait for "R" broadcast
	Q	UPDATE_Q; update for "R" broadcast
	R	IDLE; "R" broadcast
Color		Color of the AMG node 1 -> Red 2 -> Green 3 -> Blue
Resource		ID number of FU processing the AMG node, ID of IFU processing a source, or ID of OFU processing a sink

Figure 3.16. FDT file format.

CHAPTER FOUR

Case Studies and Experimental Results

4.1 Introduction

In this chapter, case studies of four algorithms are presented as a demonstration of the application capabilities of the ATAMM Simulator in studying the behavior of algorithm graphs under the ATAMM rules. These case studies are conducted and presented in a manner that typically would take the user of the Simulator through the procedural steps for creating algorithm graphs and evaluating the desired system. The first algorithm is the space surveillance algorithm. This algorithm is of particular importance because it is to be run on the ADM system. The second algorithm is the decomposed state equation for discrete linear systems. This algorithm is chosen because of its real world applicability and its recursive features. The third algorithm consists of multiple graphs with multiple sources and multiple sinks. This case study is considered to demonstrate other capabilities and features of the Simulator beyond the ADM system. The fourth algorithm is a chain that is currently being implemented on the ADM system. Results of the simulation of this algorithm are to be compared with the results of the ADM system to verify compliance with the ADM system.

A brief description of the simulation procedure using the ATAMM Simulator is presented in Section 4.2. Section 4.3 is a description of the space surveillance algorithm. Results of four case studies of that algorithm for the ideal and non- ideal cases are also presented in Section 4.3. The decomposed state equation algorithm along with its simulation results are discussed in Section 4.4. The multiple graph algorithm along with its simulation results are presented in Section 4.5. In Section 4.6, the simulated results of a three-node chain graph are compared with that of the ADM system. Finally in Section

4.7, effects of different ordering of nodes based on their priorities on the performance of the algorithm graphs are demonstrated.

4.2 Setup Procedure

The first step in using the Simulator is the creation of the algorithm graph in the Graph window. Once the algorithm graph is created, the desired architecture can be designed through the Processors and Network windows. If the injection of faults is desired, control-blocks are loaded via the PC window and the fault flag is raised. To store the simulated results for further analysis, an output file must be opened via the Simulator-Kernel window. The type of output file, the mode of operation, and the speed of the simulation process can be specified via the Simulator-Kernel window. Having all of the parameters specified and an output file opened, simulation of the algorithm graph is triggered by specifying the duration of the simulation process via the Simulator-Kernel window. The speed of the simulation can be adjusted to the desired pace at any time. Also, simulation of an algorithm graph may be paused from any window of the Simulator. Nonetheless, the simulation process may be halted only via the Simulator-Kernel or the PC windows. Abnormal termination of the simulation process, by closing the Simulator-Kernel window, will result in a loss of the output file.

The simulated results can be further analyzed via the Analyzer [18]. The desired operating point and injection rate can be selected in order to obtain the maximum throughput of the graph. Maximum throughput of the graph is achieved when the graph runs at steady state where TBIO and TBO are constant and TBO is minimal (TBO_{LB}). To arrive at the TBO_{LB}, the system is started with the value of TBO_{LB} predicted by the Design Tool [21] and the values of TBO and TBIO are observed via the Analyzer. Since the Design Tool doesn't take all the parameters of a real system into consideration, the simulated TBO_{LB} will be slightly higher. Therefore, the injection rate is increased until the steady state is reached and TBO_{LB} is found.

4.3 Space Surveillance Algorithm

The space surveillance algorithm graph, drawn in the Graph window, is depicted in Figure 4.1. The nodes describe algorithm operations and are labeled based on their priorities. Signals which are transferred from one node to another are shown as directed edges. The timing parameters of the source, sink, and nodes of this graph, consistent with the ADM implementation, are displayed in Figure A.1. However, for convenience the timing parameters are restated here. The nodes have a read-time of zero units and a write-time of three units. The process-time units of the nodes are:

Node	Process-Time
1	60
2	310
3	70
4	1240
5	100
6	1050

The write-time of the source is zero units and the read-time of the sink is three units. The injection-time of the source is initially set to the predicted value of 1250 units.

Parameters of the Processors and Network objects, based on some preliminary assumptions, are tailored to match the ADM system characteristics. These parameters are displayed in Figure A.2. The communication protocol is based on the priorities of the functional units. The test-time of the functional units is three units.

The Space Surveillance Algorithm graph is studied for two special cases. First, this graph is run under the ideal conditions as described in Section 4.3.1. Second, the graph is run under the physical constraints of the ADM system and for preselected operating points. This case is described in Section 4.3.2.

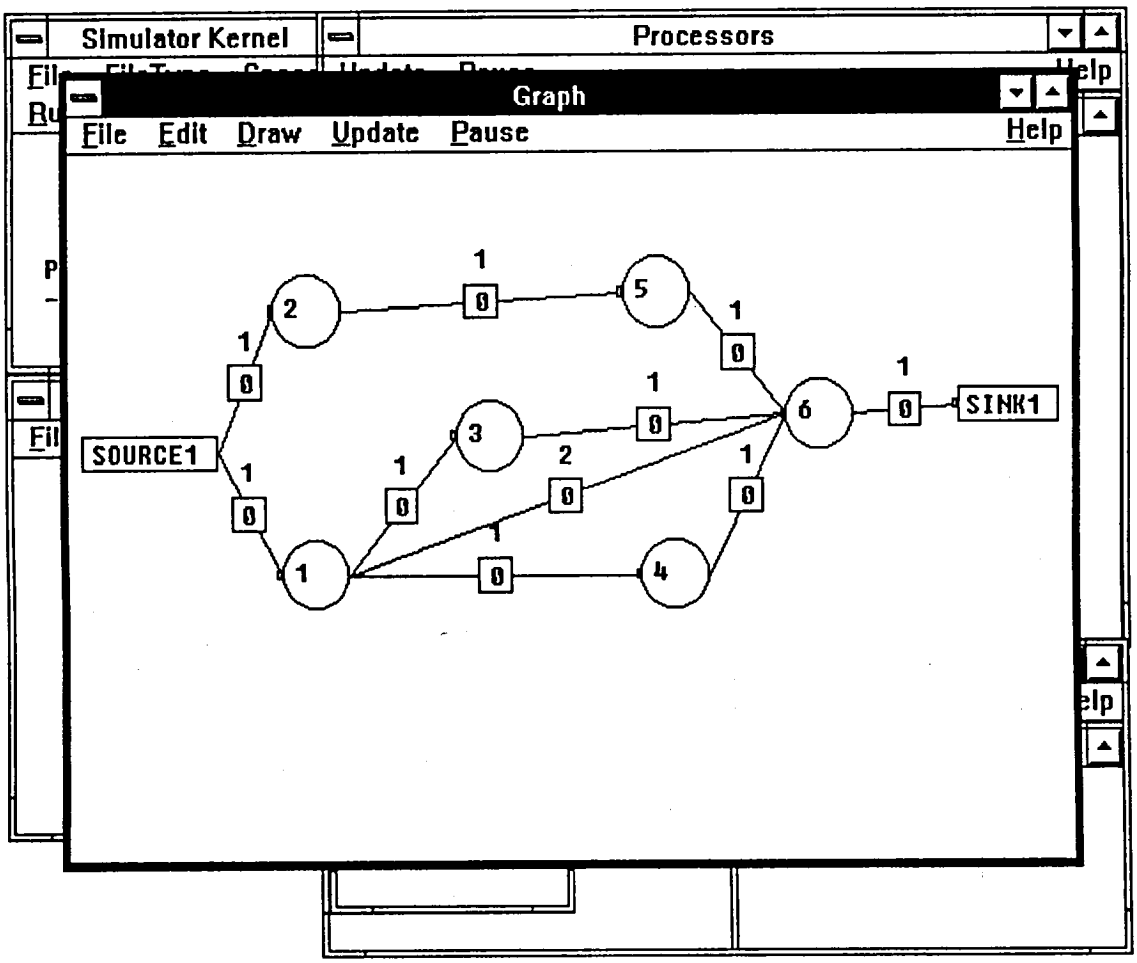

Figure 4.1. Space Surveillance Algorithm.

4.3.1 Space Surveillance Algorithm, Ideal Case

This case study is primarily conducted for validating the results of the simulation with the theoretical predictions. In this study of the Space Surveillance Algorithm, all of the parameters of the system are set to zeros. All Network parameters: grab-time, broadcast-time, wait-time, and test- time of functional units are zeroed. The read-time and write- time of the nodes are assigned to their ideal value of zero. With the parameters so specified, the only parameter that contributes to the final outcome of the graph is the process- time of the nodes. The inputs are assumed to be continuously available at the injection time so that the graph will not have to wait for input data and can run at the maximum throughput.

The ideal case is simulated for a system with four identical functional units. The functional units initially wake up in the Idle state, Figure 4.2. The current states of the functional units are reflected in the functional-unit- queues (FUQS) window of the Processor window as well as in the state diagram of every functional unit. The marking of the graph after processing a few data packets are displayed in Figure 4.3. Figure 4.4 is a display of the status of the functional units. While one of the functional units is idle, two others are processing two nodes, and yet another functional unit is undergoing a self-test. The current state of the communication channel is portrayed in Figure 4.5. The functional units contending for the channel are shown in channel-queues (CHQS) window of the Network window, Figure 4.5.

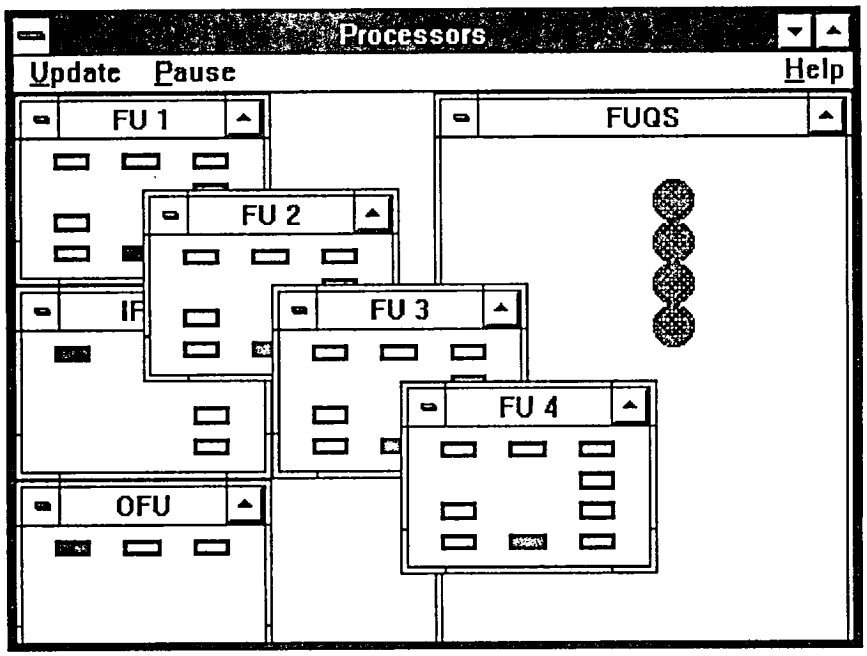

Figure 4.2. Functional Units' Initial State.

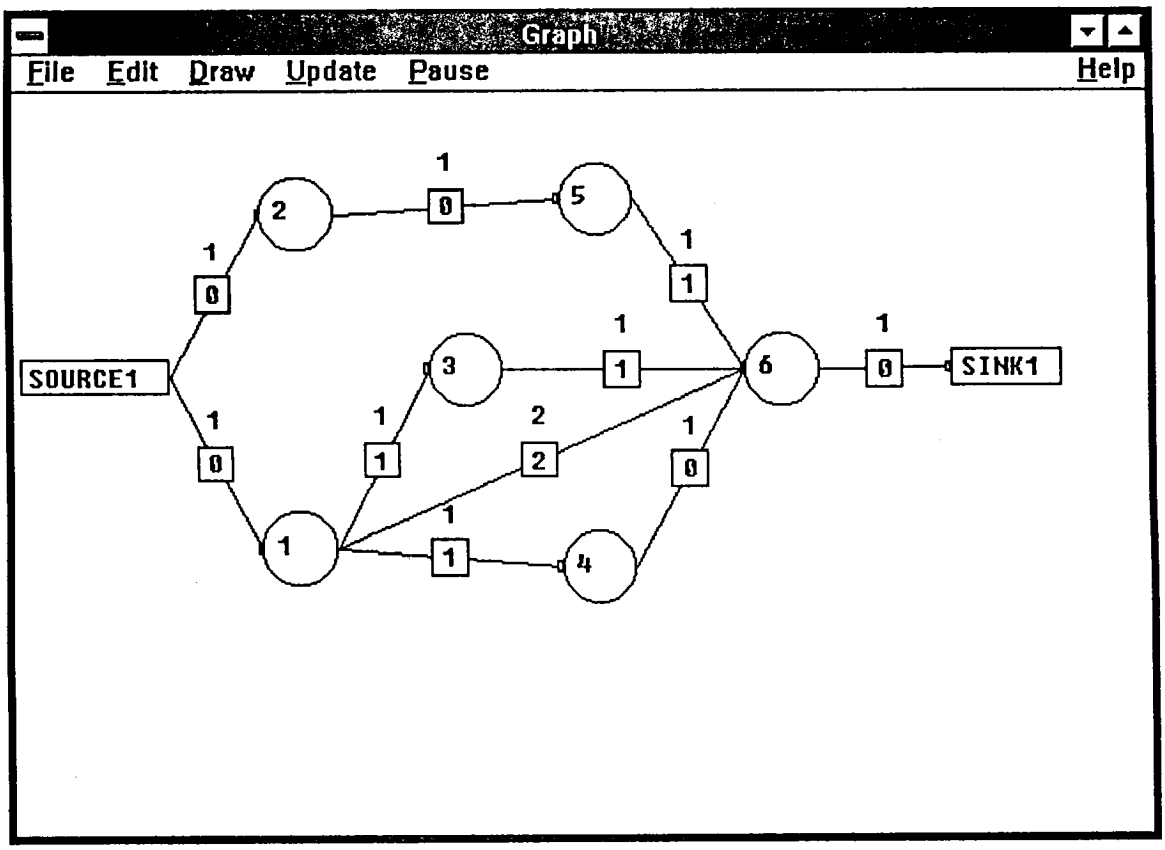

Figure 4.3. Markings of the Graph after a few Data Packets.

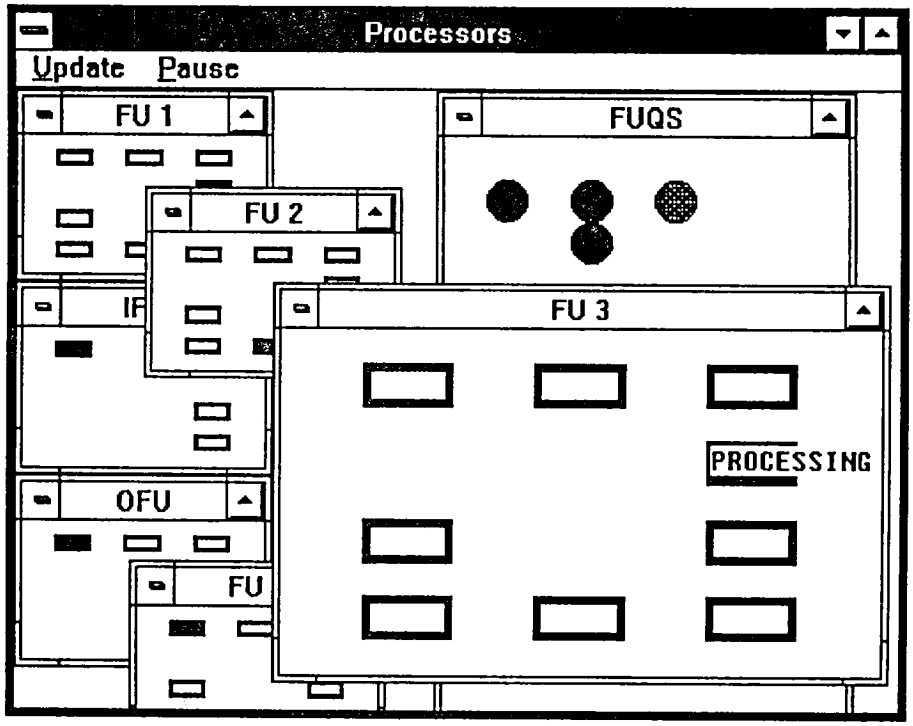

Figure 4.4. Functional Units' State after a few Data Packets.

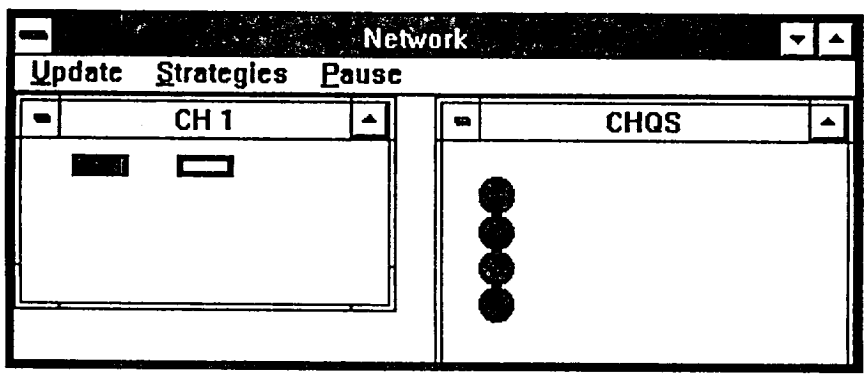

Figure 4.5. Channel's State after a few Data Packets.

The Simulator-Kernel window is shown in Figure 4.6 where the overall status of the simulation process is continuously reported. Specifically, the mode of operation, the number of clock ticks and events since the beginning of the simulation process, the number of data packets fed to the graph, and the output file and type are reported. This general information about the system and the algorithm graph provide sufficient data for monitoring the status of the simulation process. In the advent of any abnormal behavior, the state diagram of the functional units (Figures 4.7 and 3.5), the state diagram of the communication channel (Figures 4.8 and 3.9), and marking of the algorithm graph provide adequate details to pinpoint the problem. The most common abnormal behavior is chiefly due to improper initial marking of the graph, particularly the initial markings on the recursive paths.

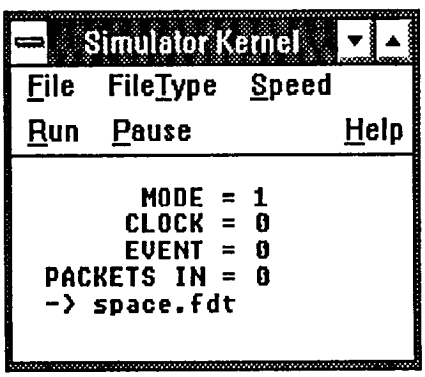

Figure 4.6. Simulator-Kernel.

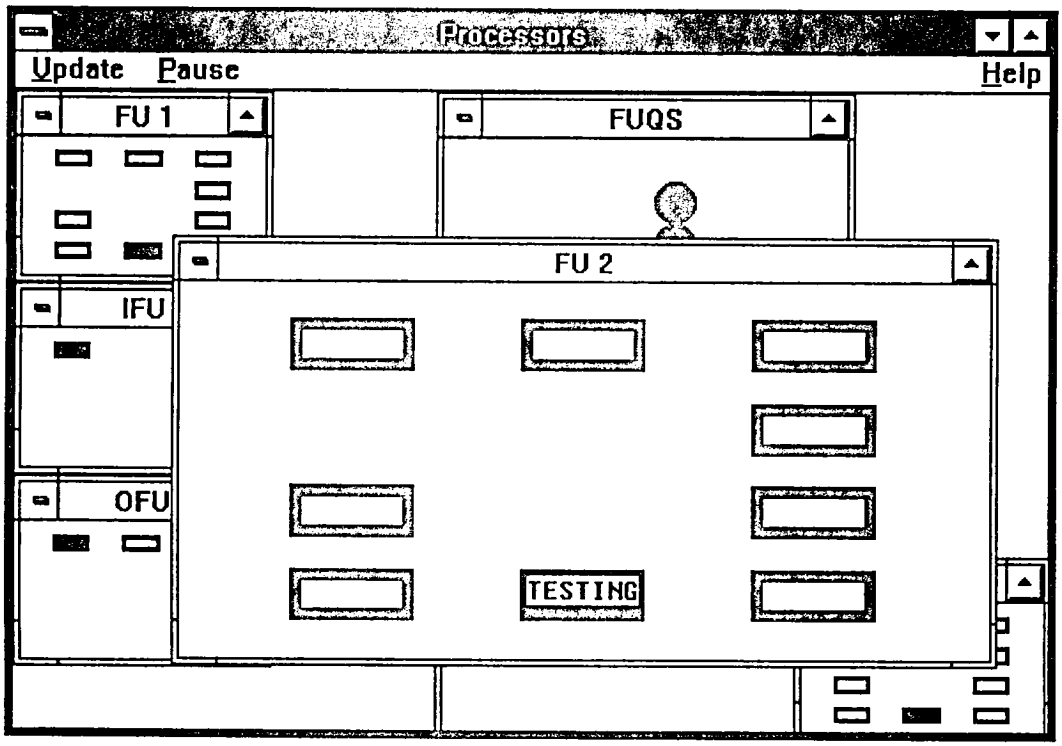

Figure 4.7. State Diagram of Functional Units.

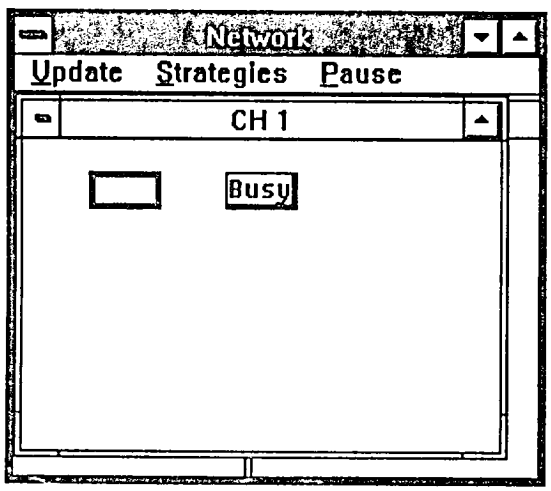

Figure 4.8. State Diagram of Communication Channel.

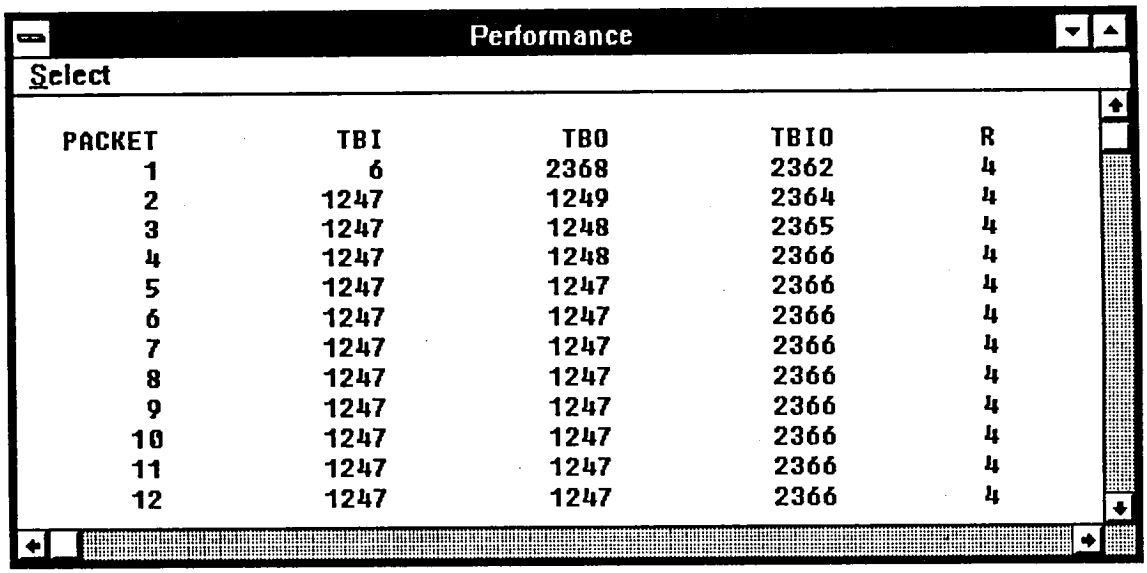

Figure 4.9. Simulated Results.

Analysis of the simulation results of this case study reveal that although the predicted TBO_{LB} is 1240 time units (process time of the largest node), the actual TBO_{LB} for this graph is experimentally found to be 1247 time units (Figure 4.9). The simulated TBO_{LB} is less than 0.5 percent more than the predicted value. The increase in the TBO_{LB} is due to the overhead imposed by the communication channel. Since this ATAMM Simulator is intended for the real systems and in real systems the communication channel is assumed to have a non- zero grab-time, every broadcast requires grabbing the channel and thus contributes to the overhead. Specifically, the time it takes a functional unit to process a node and to go back to the idle state is the process-time plus three because of the three, "F", "D", and "R" broadcasts. Also, since the system has only one communication channel, the contention for the channel has contributed to this overhead as well. Nevertheless, this overhead is not included in the non-ideal cases, because in simulating real systems, the channel is assumed to have a non-zero grab-time.

4.3.2 Space Surveillance Algorithm, Non-Ideal Cases

In this case study, the Space Surveillance algorithm is simulated under the physical constraints of the ADM system. To compare the simulated results with that of the theoretical predictions, two operating points from the performance plane of this graph [21], are chosen. For the operating point with the number of resources equal to four, the nodes have a read- time of zero units and a write-time of three units. The process-time of the nodes is the same as the ideal case. The write-time of the source is zero units and read-time of the sink is three units. The test-time of the functional units is three units. The theoretical prediction indicate that TBO_{LB} is 1247 time units. The simulated results, however, indicate that, Figure 4.10, TBO_{LB} is 1266 time units. The simulated TBO_{LB} is 1.4 percent more than the predicted value. This is primarily due to the overhead inherent in the real system, specifically the communication latencies. The other

factor that contributes to this overhead is the communication channel contention where more than one functional unit contend for the communication channel.

For the operating point with the number of resources equal to two, the algorithm graph is modified to improve performance. The modified algorithm graph is shown in Figure 4.11 where the queue size of some edges is increased and two control edges are added. The control edges are added from node 3 to node 2 and from node 4 to node 3. Analysis of the modified algorithm graph indicates that for the same injection-time as previous case, TBIO should increase. This arrangement corresponds to a horizontal move in the performance-plane of this algorithm [1]. The simulated results, given in Figure 4.12, indicate a slight increase in the value of TBO at the steady state.

Graceful degradation of operation of an algorithm graph in real-time systems is modeled by the ATAMM model and is reflected in the performance-plane of the algorithm graph. As the number of available resources changes, new operating points are selected and the injection rate is adjusted and/or the graph is appropriately modified (Section 3.10). The desired operating points and the required modifications are pre-selected by the user and are conveyed by an CTL file.

The above two cases are combined to show the graceful degradation of the operation of an algorithm graph in real- time. To show the transition of the operation of the algorithm graph from one operating point, operating point equal to 4, to another, operating point equal to 2, two of the functional units are removed from the system after reaching the steady-state operation. The new injection rate and information about modifications of the graph are shown in Figure A.3. For this case study, the CTL file of Figure A.3 must be loaded via the PC window and the FAULT option must be selected. As expected, the simulated results shown in Figure 4.13 indicate that the TBO and TBIO at the steady-state and for operating points equal to 4 and 2 are the same as the previous case studies. Figure 4.14 shows the reduction in the number of resources. The modified graph is the same as in Figure 4.11.

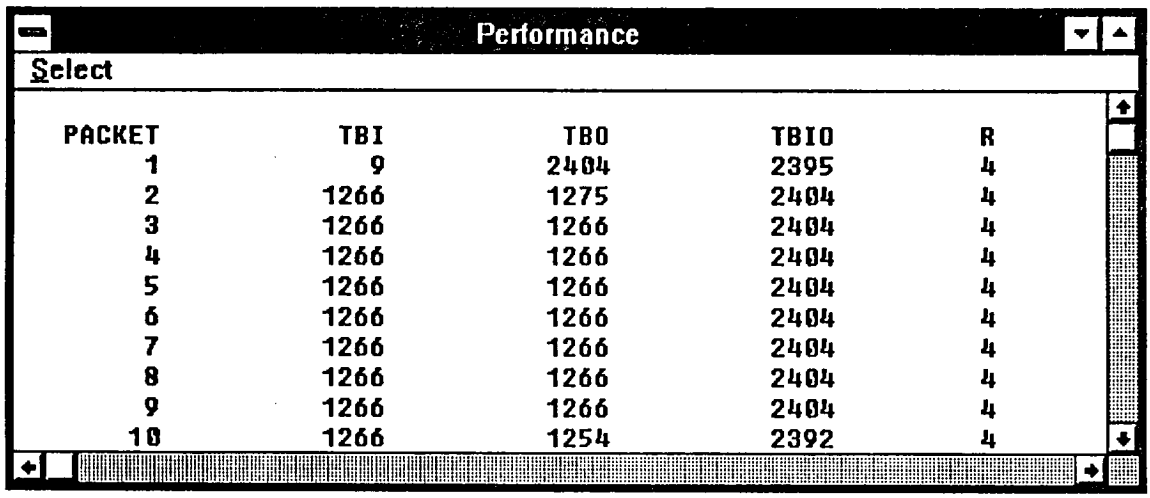

Figure 4.10. Simulated Results.

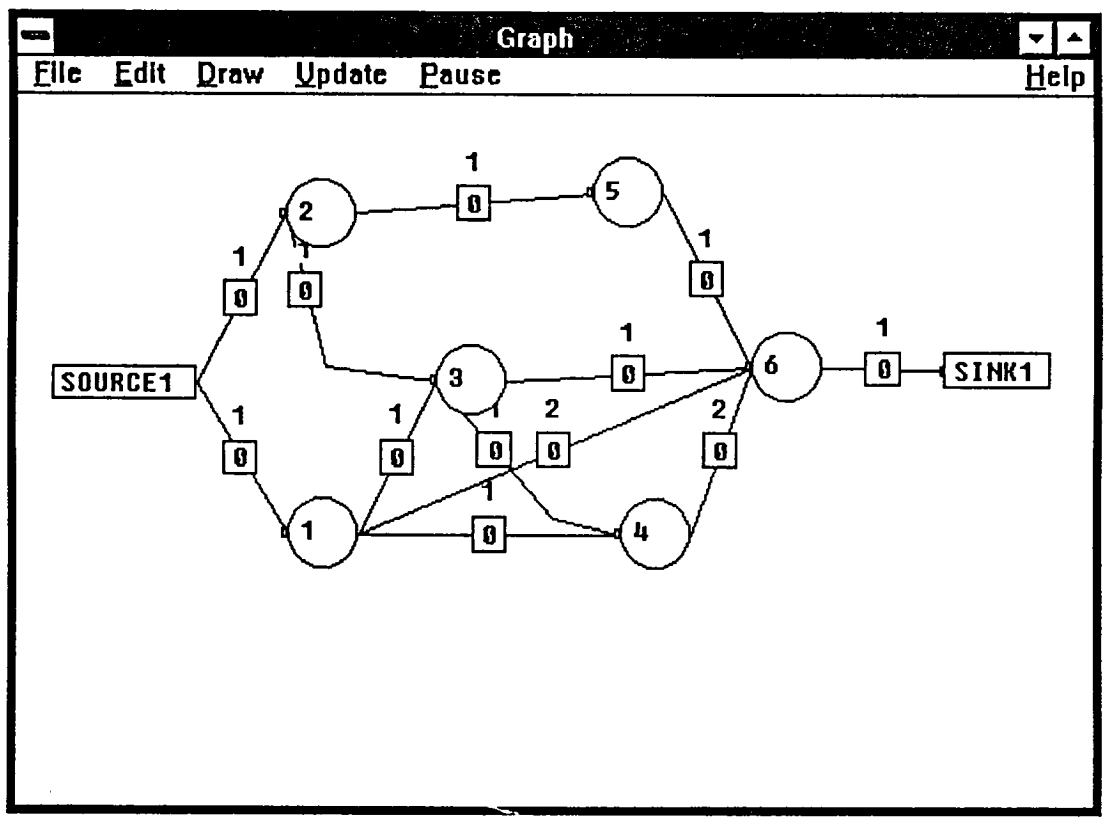

Figure 4.11. Space Surveillance Algorithm.

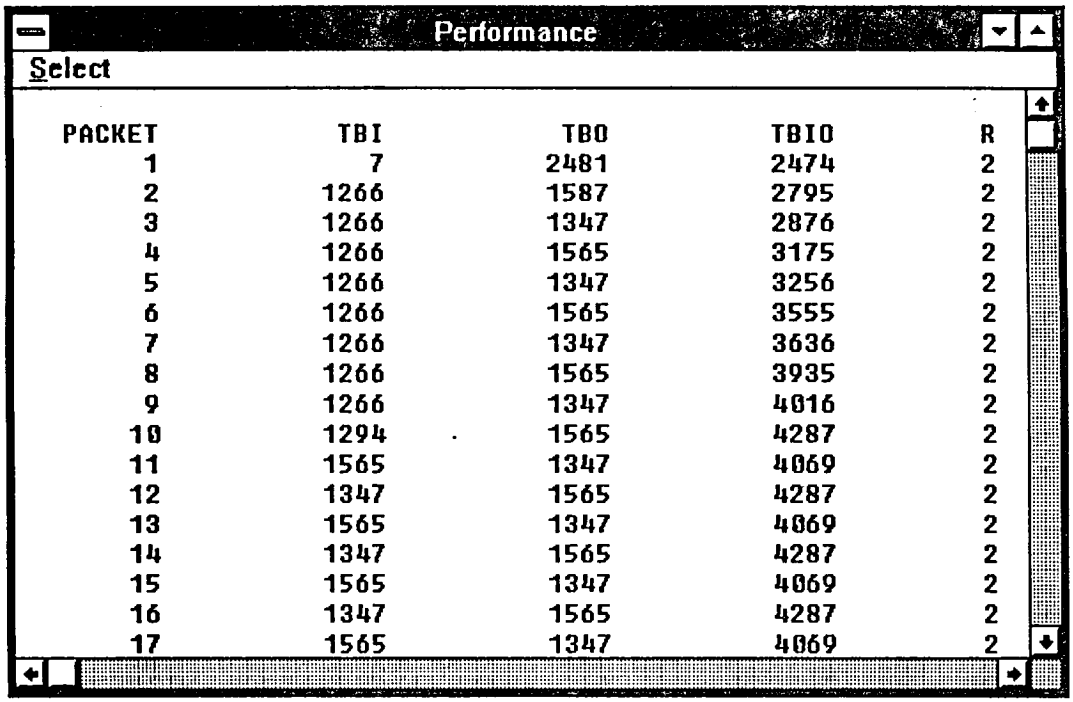

Figure 4.12. Simulated Results.

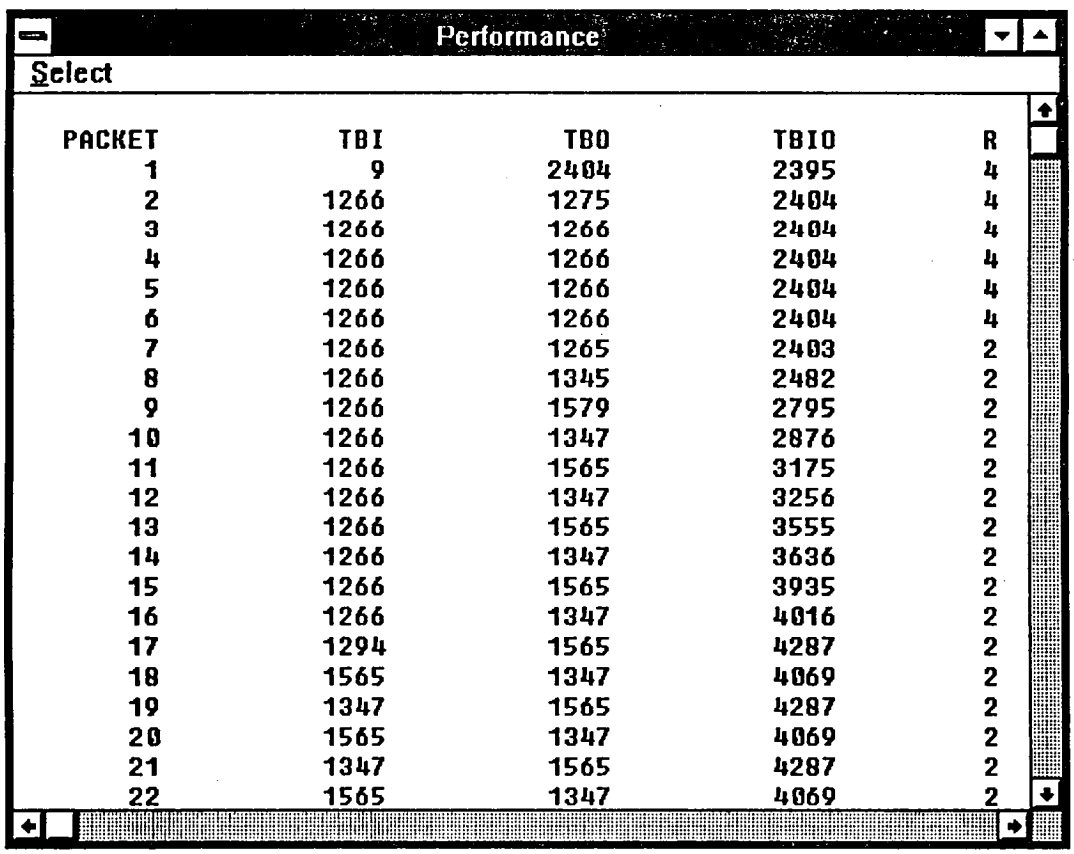

Figure 4.13. Simulated Results.

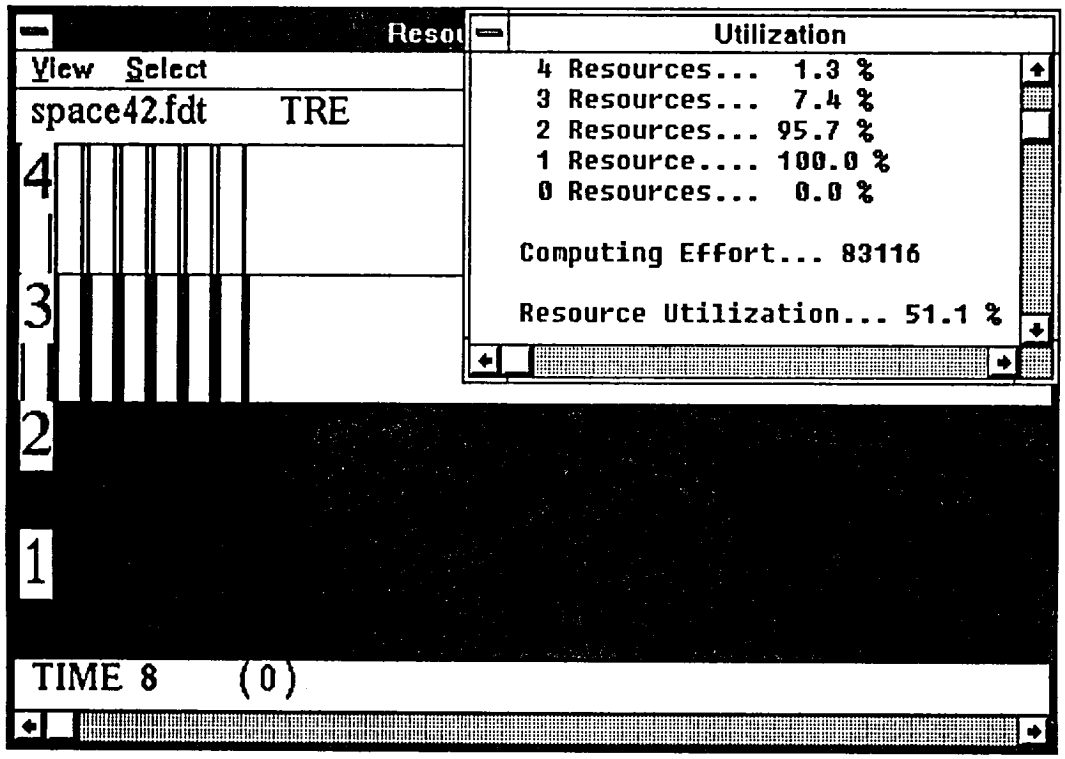

Figure 4.14. Simulated Results.

4.4 Decomposed State Equation

Consider the problem of computing the output of a discrete linear system given a sequence of inputs to the system. Let the system be described by the state equation

$$X(K+1) = A\,X(K) + B\,U(K+1),$$

and output equation

$$Y(K+1) = C\,X(K+1)$$

where X is a p-vector, U is an m-vector, and Y is a r-vector. The primitive operations are defined as matrix multiplication and vector addition.

For the purpose of this case study, the state equation is decomposed so that the node times are reduced and parallel execution of nodes is possible [11]. This decomposition lowers the value of TBO, due to reduced node time, and thus increases throughput. The decomposition of the state equation is performed as follows,

$$\begin{bmatrix} X_1(K+1) \\ X_2(K+1) \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} X_1(K) \\ X_2(K) \end{bmatrix} + \begin{bmatrix} B1 \\ B2 \end{bmatrix} (U(K+1)),$$

and

$$(Y(K+1)) = (C_1, C_2) \begin{bmatrix} X_1(K+1) \\ X_2(K+1) \end{bmatrix}.$$

The AMG representing the decomposed state equation is drawn in the Graph window of the ATAMM Simulator and is shown in Figure 4.15. The nodes and the edges are labeled in accordance with the above two equations.

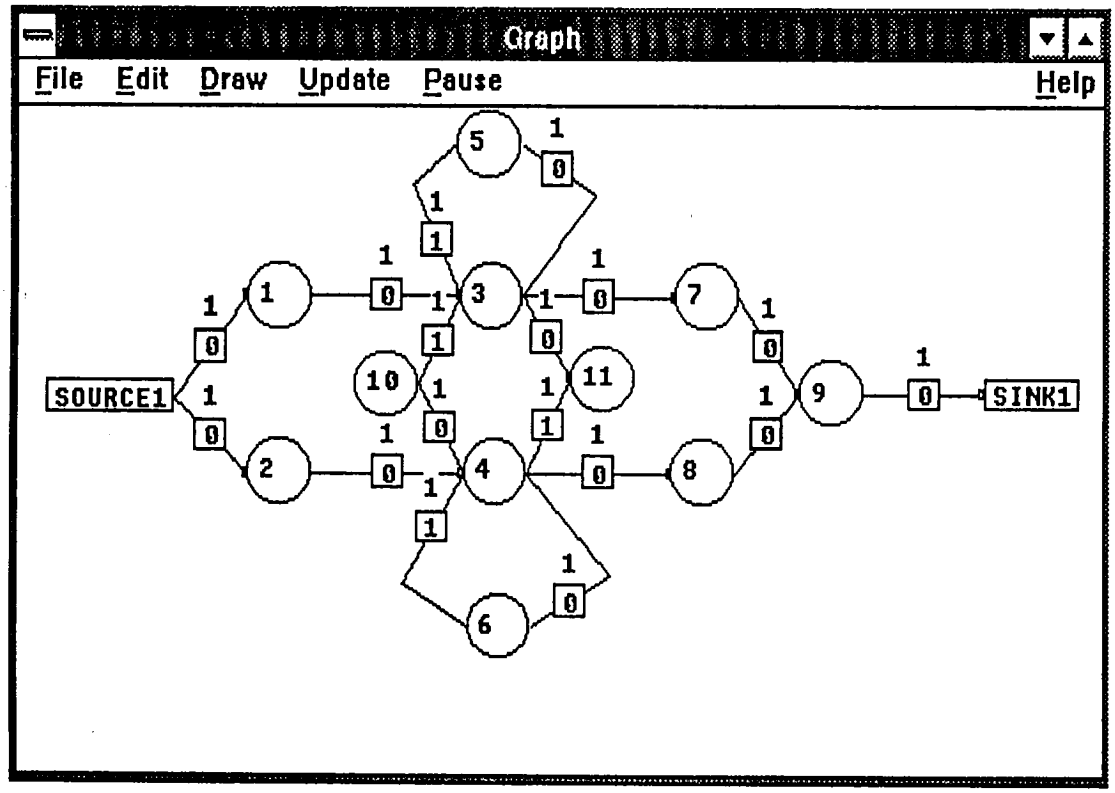

Figure 4.15. Decomposed State Equation.

To compare the simulated results with that of the theoretical predictions, an operating point equal to 8 from the performance plane of this graph [21] is chosen. For this operating point, the parameters are specified as shown in Figure 4.16. The theoretical prediction indicates that the TBO_{LB} is 1000 time units. The simulated results indicate that, Figure 4.17, TBO_{LB} is 1158 time units. The theoretical predictions and the simulated findings of all graphs are tabulated and shown in Table 4.1. As is indicated in Table 4.1, the overhead for this graph is quite noticeable, about 15 percent. As stated earlier, the amount of overhead also depends on the contention for the communication channel. Contention for the communication channel is a direct product of the number of parallel paths in the graph as well as the number of nodes that can concurrently fire. In

other words, contention is directly proportional to the amounts of parallel and pipeline concurrency (Section 2.2).

The study of the AMOS's state diagram reveals that the "R" broadcast accounts for one third of total broadcasts and thus one third of communication overheads. To study the contribution of different parameters to the overhead, the Decomposed State Equation graph is simulated after eliminating the test-time of the functional units, as shown in Table 4.1. As expected, the overhead is slightly reduced. The amount of reduction, of course, is directly proportional to the value of test-time. This study is carried on even further where by combining the "R" broadcast in the "D" broadcast and eliminating the test-time, the Test state of the AMOS state diagram is bypassed. The simulated results, tabulated in Table 4.1, signify over 30 percent increase in performance. These results are indications of the amount of overhead contributed by the third broadcast and continuous testing of the functional units. However, if it is not necessary to test the functional units continuously, or if it is desired to test the functional units every so often, the current state diagram of the AMOS needs to be modified to provide the required flexibility.

A new modified state diagram for AMOS is proposed where the capability for deciding to conduct a self test by the functional units is provided, Figure 4.18. With the proposed state diagram, it is possible to completely isolate the overhead associated with the Test state of AMOS or to test the functional units at the desired intervals. Thus, it is now possible to manage the tradeoff between performance and fault tolerance.

MODE1

NumofFUS	8
Protocol	1
Grab_Time	3
BDCT_Time	2
Test_Time	2
Updt_Time	2
Wait_Time	0

Node	Process-Time	Read-Time	Write-Time
1	500	0	0
2	500	0	0
3	200	0	0
4	200	0	0
5	800	0	0
6	800	0	0
7	400	0	0
8	400	0	0
9	150	0	0
10	800	0	0
11	800	0	0
Source	-	-	0
Sink	-	0	-

Figure 4.16. STP file and timing parameters for the Decomposed State Equation.

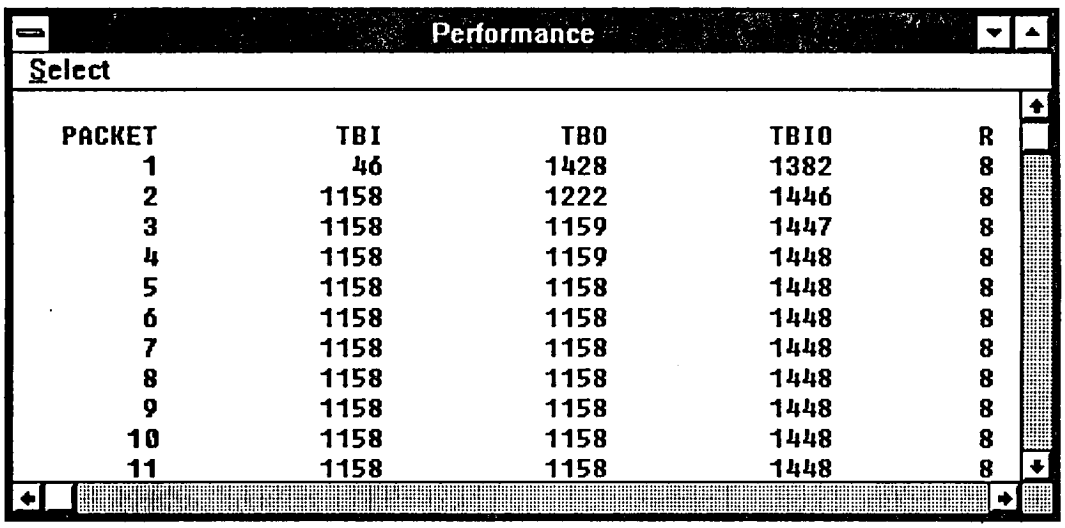

Figure 4.17. Simulated Results.

Algorithm	Predicted TBO_{LB}	Sim'ed TBO_{LB}	Sim'ed TBIO	R	Percentage Over-head	Comment
Space	1240	1247	2366	4	0.56	Ideal case.
Space	1247	1266	2404	4	1.52	
Space	1439	1456	2913	2	1.18	
State Eq.	1010	1158	1448	8	14.65	
State Eq.	1010	1154	1451	8	14.25	test-time = 0.
State Eq.	1008	1102	1386	8	9.32	Bypassing the Test state of AMOS.

Table 4.1. Predicted and simulated results of the case studies.

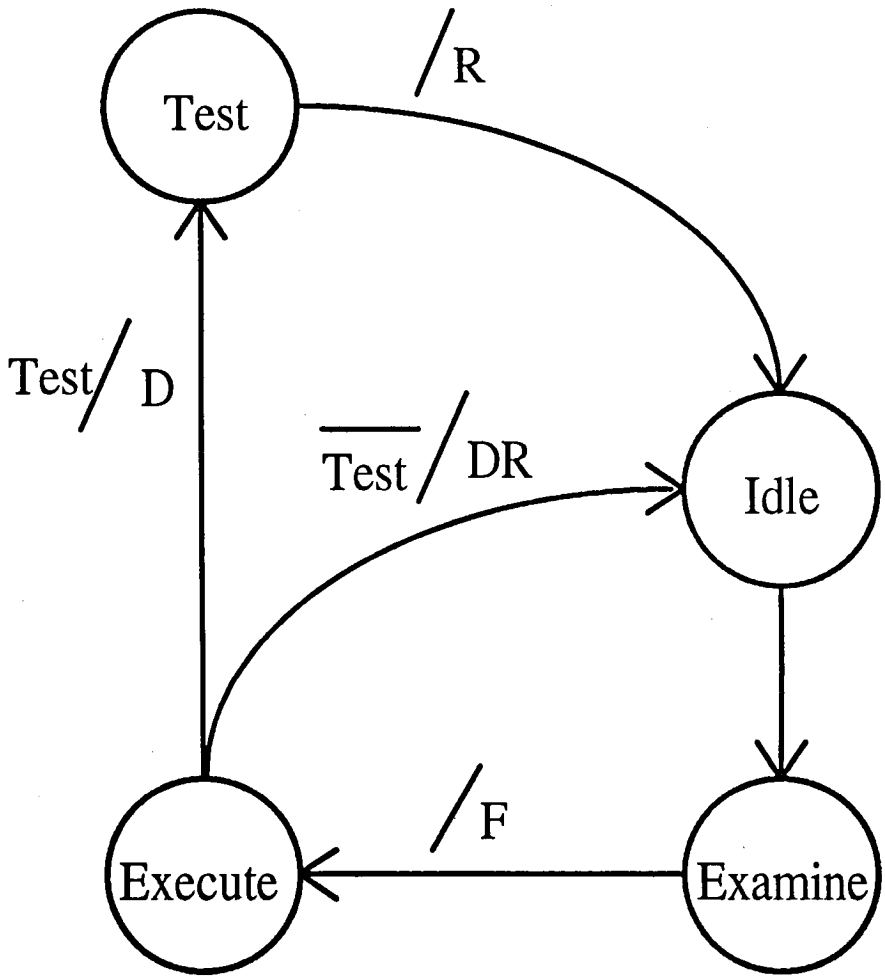

Figure 4.18. Modified AMOS State Diagram.

4.5 Multiple Algorithm Graphs with Multiple Sources and Sinks

This case study is considered to demonstrate other capabilities and features of the Simulator beyond its ADM system compatibility. The ATAMM Simulator is capable of concurrently simulating multiple independent graphs as well as graphs with multiple sources and sinks. Since all sources, sinks, and nodes have priorities, the multiple graphs created in the Graph window are inherently prioritized. Therefore, when creating the multiple graphs, the order in which the nodes are created and connected to each other defines the priorities of the nodes and hence a priority relationship among the graphs. In

graphs with multiple sources and/or sinks, the sources have independent injection rates. The sources independently contribute to the overall performance of the graph. By controlling the injection rate of the independent sources, it is possible to find the overall TBO_{LB} of the graph.

Two graphs are considered for this case study. The graphs are drawn in the Graph window of the Simulator and are shown in Figure 4.19. The first graph has two sources and two sinks. The second graph is a three-node chain with a single source and a single sink. Since the source, sink, and nodes of the second graph have higher ID's than the first graph, the first graph has higher priority than the second graph. The injection rate of the sources of the first graph are 420 and 220 time units, respectively and the injection rate of the second graph is 420 time units. The process-time of all nodes of both graphs is 100 time units. The read-time and write- time of all nodes and sinks of both graphs is 5 time units.

Analysis of the simulated results indicate that simultaneous simulation of both graphs is faithfully carried out. The results also indicate that the individual graphs are simulated in compliance with the ATAMM rules.

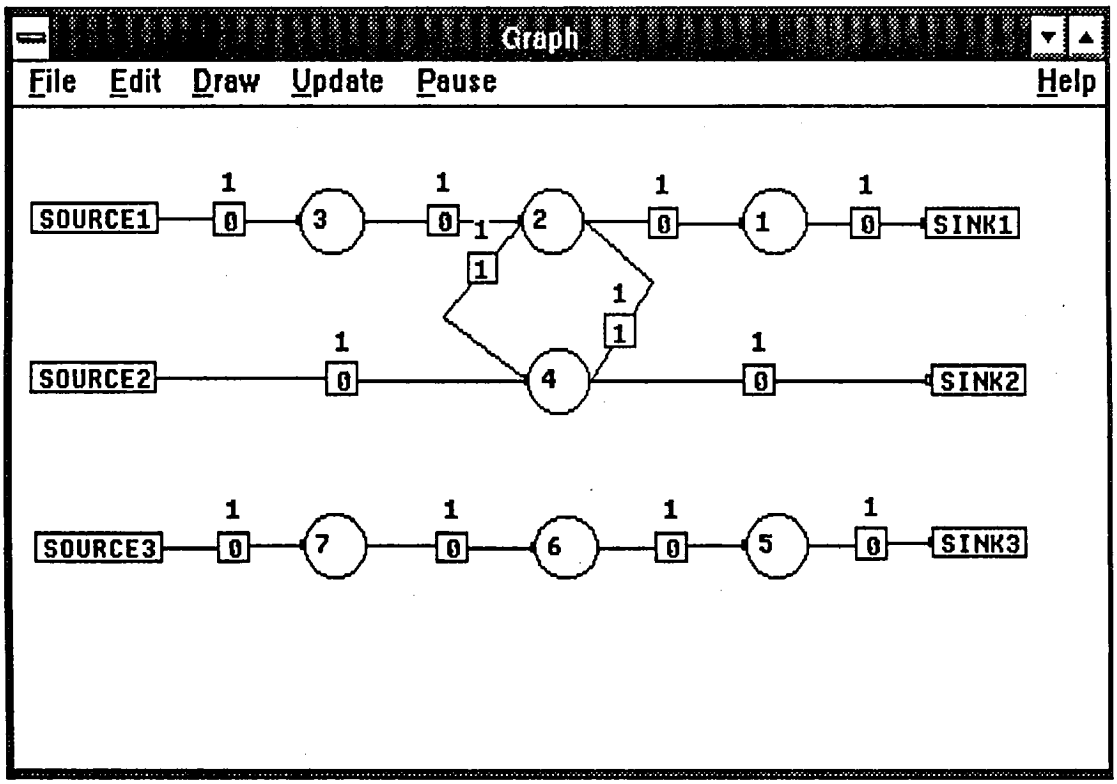

Figure 4.19. Multiple Graphs with Multiple Sources and Sinks.

4.6 Chain Graph

An AMG consisting of a three node chain is currently being implemented on the ADM system. The graph is shown in Figure 4.20. This graph is considered for this case study and the results of the simulation of this graph will be compared with that of the ADM system. Comparison of the results demonstrates the compliance of the ADM system with the ATAMM model as represented by the Simulator taking all of the parameters of the ADM system into account.

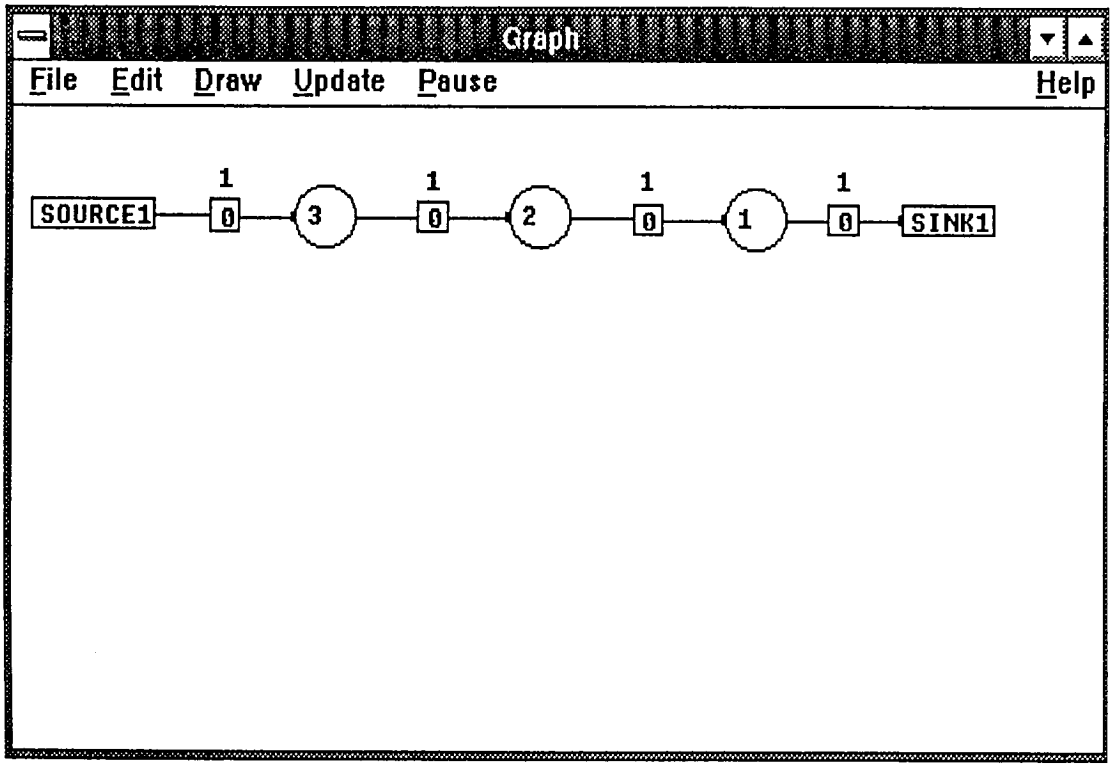

Figure 4.20. Three Node Chain Graph.

Another aspect of this case study is the demonstration of some of the fault tolerant features of the Simulator as well as the ADM system. Through this case study injection of faults, changes in the operating points, and the use of the control blocks are demonstrated.

The initial operating point chosen for this graph is equal to 3. For this operating point, the parameters are specified as shown in Figure 4.21[3]. The theoretical predictions indicate that, for an operating-point equal to 3, the TBO_{LB} is 2489 time units. The simulated results indicate that, Figure 4.23, TBO_{LB} is 2527 time units. However, the experimental results of the ADM system indicate that TBO_{LB} is 2761 time units. The theoretical predictions, simulated findings, and experimental results of this graph for all

[3] To compare the simulated results with the results of the ADM system, the data attained from the ADM system are normalized.

operating points are tabulated and shown in Table 4.2. The CTL file for this case study is shown in Figure 4.22. As the CTL file indicates, the functional units 2 and 3 will malfunction when data packets 7 and 15, respectively, are injected by the source. The defective functional units will remove themselves from the system (Section 3.10) immediately if they are in the Test state or upon entering the Test state. As seen in Figure 4.23 the fault injection caused the functional units 2 and 3 to be removed from the system at the completion of data packets 4 and 12 respectively.

A study of Table 4.2 indicates that the simulated TBO_{LB} is about 8 percent less than the experimental TBO_{LB} achieved by the ADM system for an operating-point equal to 2. The discrepancy between the simulated and the experimental results is due to two reasons. First, the timing parameters associated with the nodes and the system were extracted from the results of the ADM system and averages of the extracted values were used in the simulation process. Second, there is certain randomness associated with the communication channel protocol adapted in the current version of the ADM system.

The changes in the number of resources for this case study are shown in Figure 4.24 (the upper portion of this and the next Figures is that of the simulated results while the bottom portion is that of the ADM system). The simulated and experimental resource utilizations are shown in Figure 4.25. The functional unit activities are shown in Figure 4.26. As is evident in Figures 4.24, 4.25, and 4.26, the simulated and experimental results are highly comparable. The consistency in the simulated and experimental results is an indication of the compliance of the ADM system with the ATAMM rules.

MODE1
NumofFUS 3
Protocol 1
Grab_Time 4
BDCT_Time 3
Test_Time 60
Updt_Time 0
Wait_Time 0

Node	Process-Time	Read Time	Write Time
1	1556	3	3
2	843	3	3
3	2480	3	3
Source	-	-	3
Sink	-	3	-

Figure 4.21. STP file and timing parameters for the Chain Algorithm.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7 0 0 2
15 0 0 3
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
3
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Figure 4.22. The CTL file for the Chain Graph.

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Figure 4.22. (continued) The CTL file for the Chain Graph.

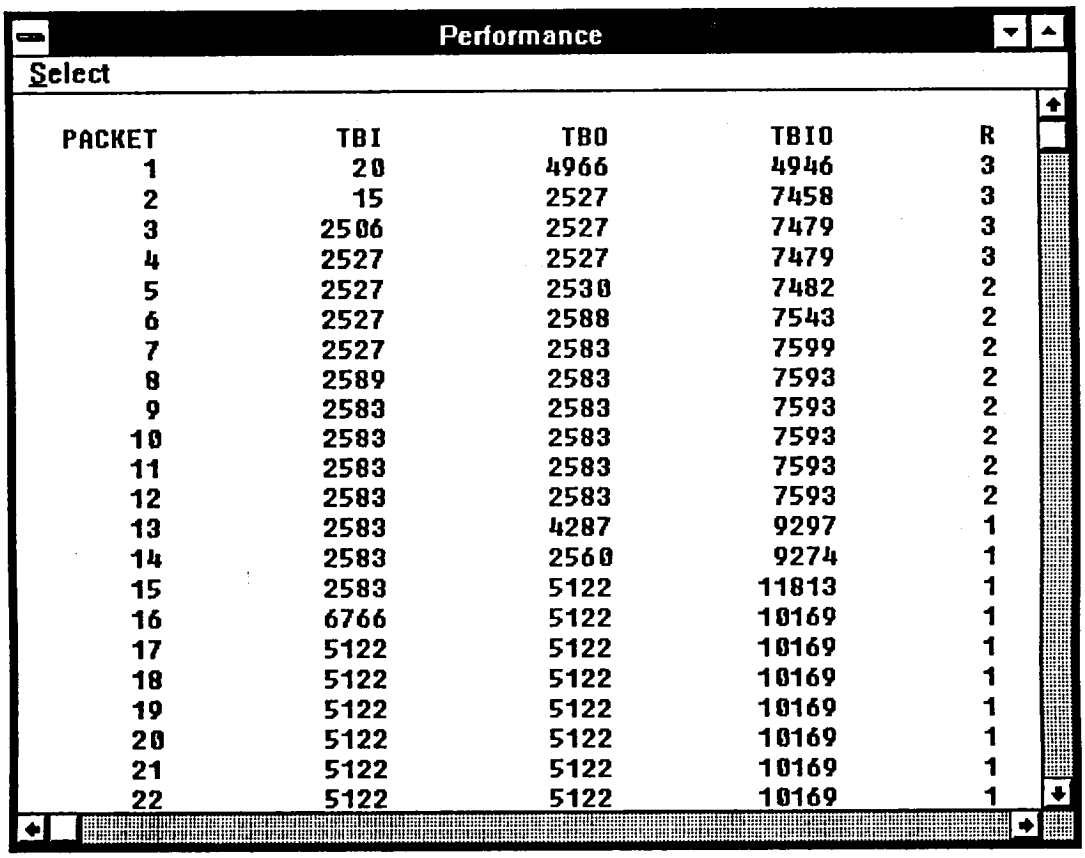

Figure 4.23. Simulated Results Corresponding to Output Data Packets.

Theoretical TBO_{LB}	(Design Tool) Predicted TBO_{LB}	Simulated TBO_{LB}	(ADM) Experimental TBO_{LB}	Operating-Point R
2489	2489	2527	2761	3
2489	2489	2583	2764	2
2489	4903	5122	5338	1

Table 4.2. Results of the Chain Graph case study.

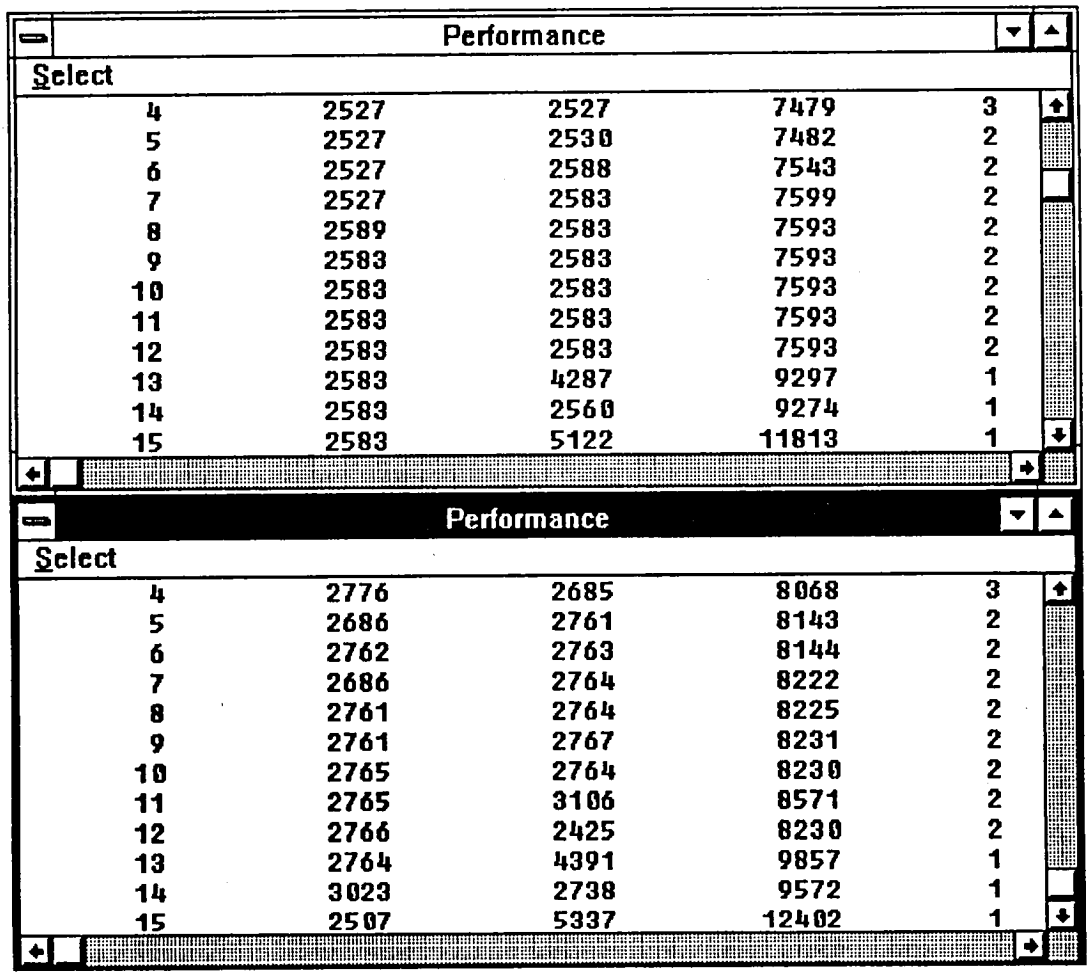

Figure 4.24. Simulated and ADM Results.

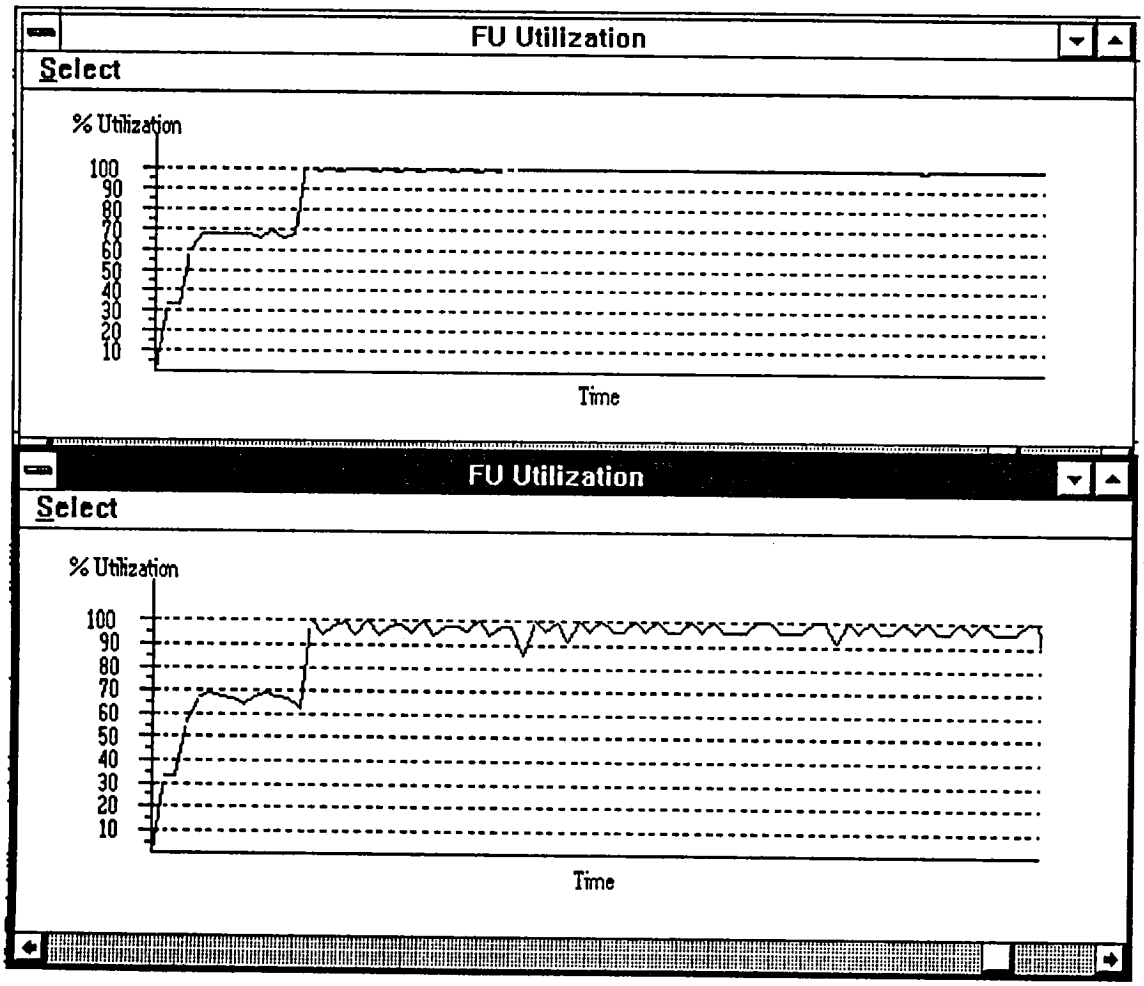

Figure 4.25. Simulated and ADM Results.

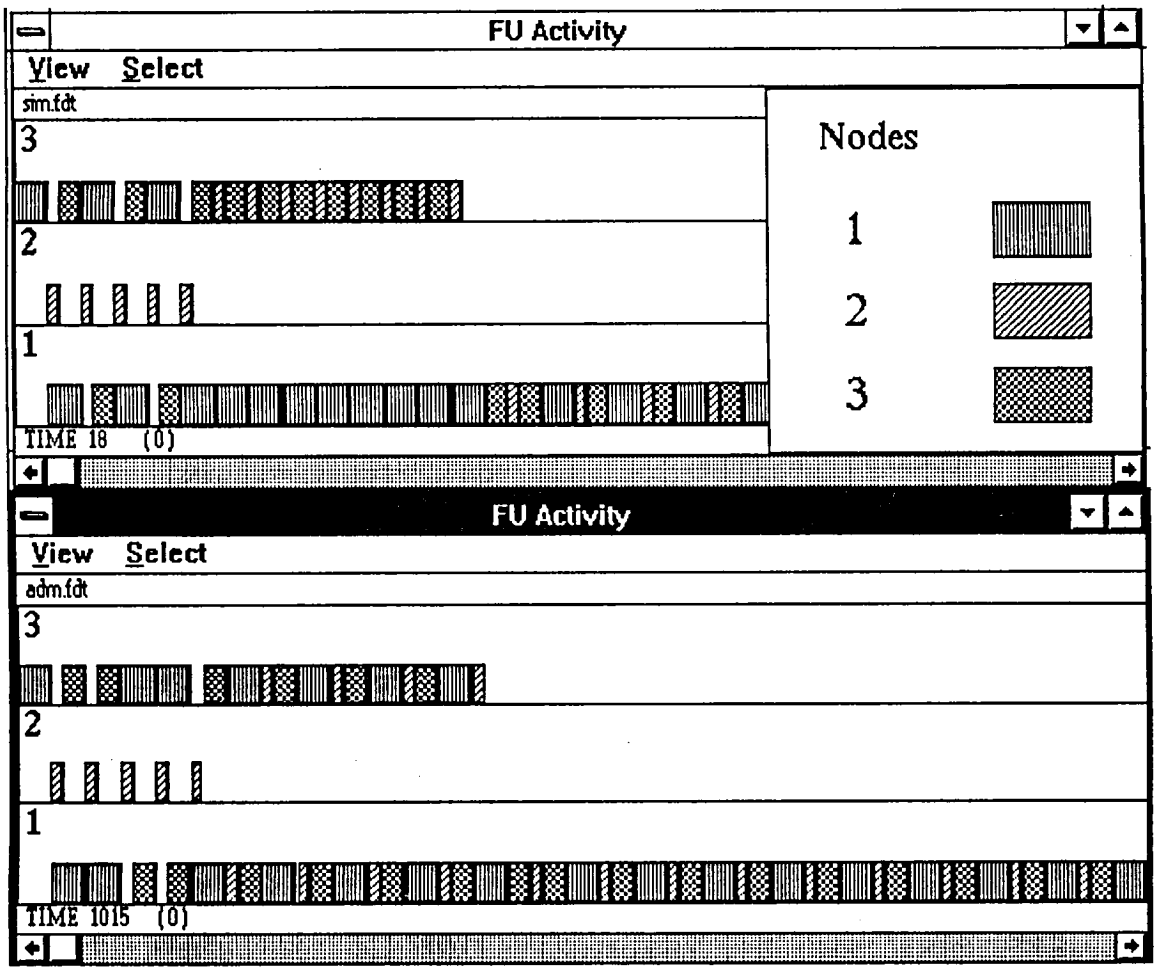

Figure 4.26. Simulated and ADM Results.

4.7 Experimental Results

Effects of different orderings of nodes based on their priorities on the performance of the algorithm graphs are briefly discussed in Section 4.7.1. However, a formal proof of these findings requires an extensive study of all possible cases and thus is beyond scope of this thesis.

4.7.1 Effects of Node Priority on Performance

A three-node chain graph is considered to show the effect of different orders of priorities of nodes of a graph on the performance of that graph. For this study two extreme cases of orderings of the nodes are considered. First, the nodes are ordered in ascending order from the source to the sink with the lowest number having the highest priority. The read and write time of the nodes, write time of the source, and read time of the sink are 5 time units while process time of the nodes are 100 time units. For an operating-point equal to 1 and an injection-time of zero, the simulated TBO and TBIO are 459 and 1189 time units, respectively. Second, the nodes are ordered in the descending order from source to sink. The simulated TBO and TBIO are 459 and 876 time units, respectively. Primary analysis of the simulated results indicate that for the case where the nodes are in ascending order 1) it takes longer to reach the steady state, Figure 4.27 (the upper portion of this and next Figures is that of the ascending ordering of the nodes), 2) TBIO is longer, because more data packets are fed into the graph during one TBIO, Figure 4.28, and 3) at steady state, TBO is the same as for descending orderings of the nodes. However, for an operating-point equal to 2, the results are less dramatic. For an operating-point equal to 3, the simulated TBO and TBIO are 232 and 804 time units, respectively, for both cases. Therefore, if there are as many functional units as nodes in the graph, then different orderings of priority of the nodes do not increase the TBIO.

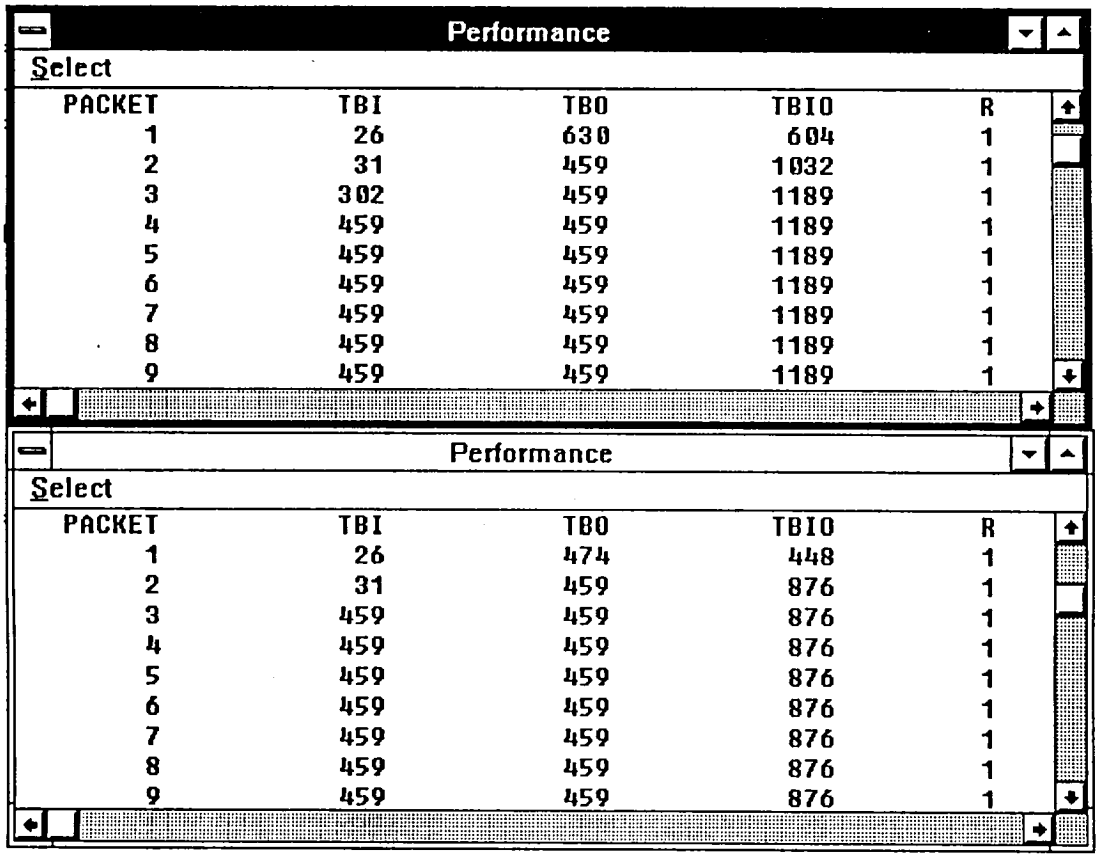

Figure 4.27. Simulated Results.

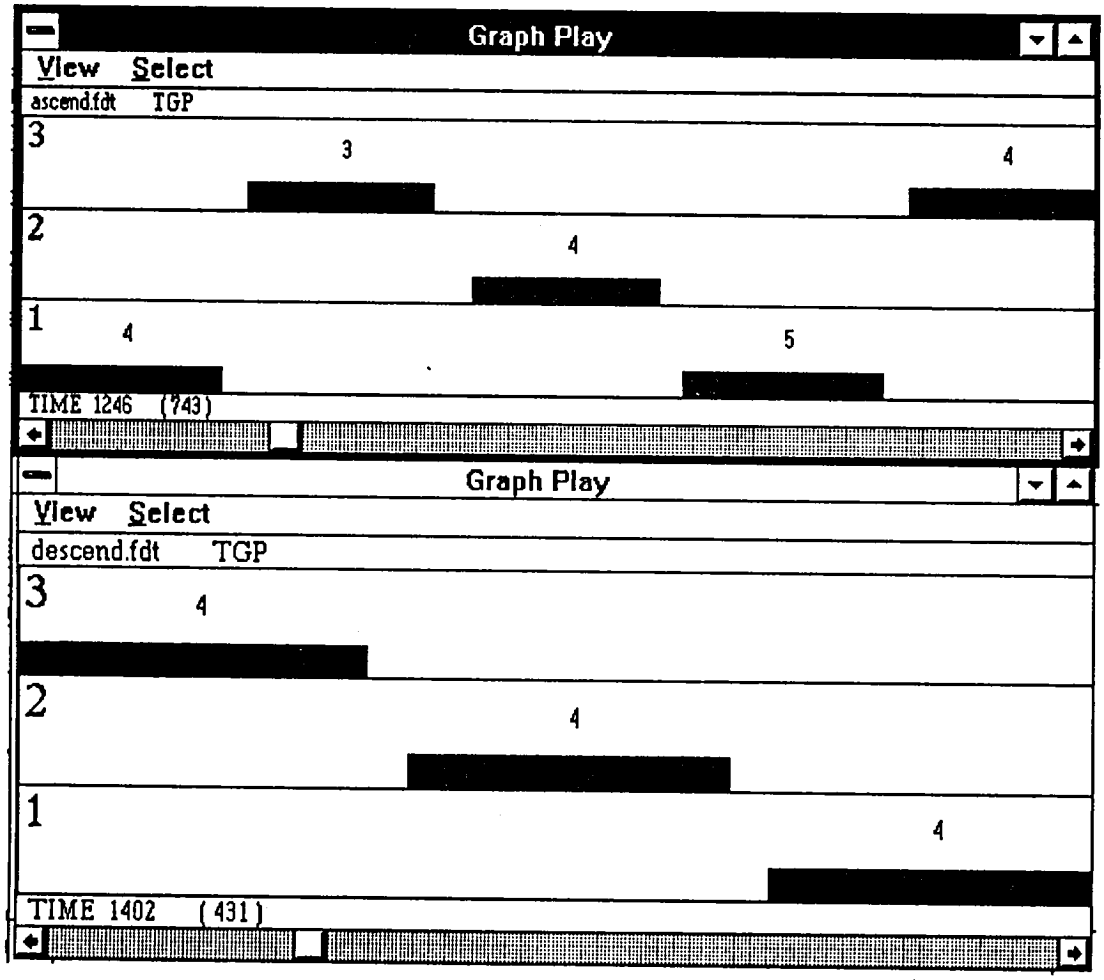

Figure 4.28. Graph Play in One TBIO.

CHAPTER FIVE

Conclusion

5.1 Summary

The ATAMM model, a new Petri net based model developed by the researchers at the Old Dominion University, provides the analytical means to integrate algorithm data flow with data- flow architecture. The ATAMM model provides a description of the data and control flow necessary to specify the criteria for predictable execution of an algorithm by a data flow architecture. The ATAMM model also provides the means to investigate different algorithm decompositions without having to consider the hardware. Once the intended hardware is selected, the model can be used to match the algorithm requirements with the hardware capability in order to achieve optimum performance.

A simulation program was developed and is presented in this thesis in order to productively aid the user of an ATAMM based distributed-processing system in the evaluation and design process of a particular system to determine its optimum performance. The software is referred to as the ATAMM Simulator. The Simulator provides the means to permit an architecture-independent study of behavior, performance, and reliability of a system without having to build a hardware prototype. Results of execution of an algorithm by the Simulator are comparable to the results of the ADM system. The Simulator is able to assist with the development of ATAMM based architectures and the investigation of theories concerning the ATAMM model. In order to ease and facilitate user interactions, this user-friendly software was developed within a window environment. Utilizing a window environment permits the Simulator to run concurrently with other software applications (such as the Analyzer). The window environment also permits one to view all of the Simulator displays simultaneously.

As a demonstration of the application capabilities of the ATAMM Simulator, case studies were performed on four different algorithms. First the Space Surveillance Algorithm was considered where for the ideal case, compliance of the simulated results with the theoretical predictions was verified. Second, the Decomposed State Equation for discrete linear systems was considered to exhibit the effect of presence of recursive paths in the algorithm graph. Third, multiple algorithm graphs with multiple sources and sinks were considered to demonstrate other capabilities and features of the Simulator beyond the ADM system. The last algorithm considered was a three-node chain graph. Simulated results of the three-node chain graph were shown to be comparable to that of the ADM system.

The Simulator's capability to incorporate attributes of a generic system was also exhibited. These Simulator features are essential in order to productively study an ATAMM-based system. The Simulator was used to determine the effects of overhead associated with a real system on the performance of the algorithm graphs. Without the Simulator, such investigation of system overhead would be difficult. The Simulator also was used to determine performance of the Space Surveillance Algorithm and the three-node chain graph in a degraded mode. For these two algorithms, injection of faults, changes in the operating-points, and the use of the control blocks were demonstrated. The results of these case studies were presented through a set of ATAMM Simulator window displays. Finally, it was shown that the order of priorities of the nodes in an algorithm graph is very important. For low TBIO, high priority nodes must be closer to the sink and low priority nodes closer to the source.

5.2 Topics for Future Research

Current research is concentrated in extending the capabilities of the ATAMM Multicomputer Operating System, and thus expanding the problem domain of ATAMM. The enhanced AMOS is to be implemented in the Generic VHSIC Spaceborne Computer (GVSC), a spaceborne, four-processor breadboard which is also based on the 1750A instruction set architecture. In this regard, the ATAMM model is being generalized to permit multiple concurrent instantiations of selected graph nodes. Also, the simultaneous play of multiple graphs, each having a distinct source node and sink node, is being developed. Three separate strategies for implementing multiple graphs are being considered. These strategies are referred to as the parallel execution strategy, the time multiplexing strategy, and the priority interrupt strategy. The different strategies are selected to address classes of problems which arise in real- time applications. Efforts are also being made to incorporate the features of fault-tolerance and branching in the ATAMM model.

Future research could involve the inclusion of more classes of faults and better fault detection and recovery strategies. Enhancements to ATAMM to include graphs with multiple sources and sinks and graphs with variable node times should also be investigated. The ATAMM model should also be enhanced to incorporate heterogeneous architectures as well as systems with multiple communication channels. These enhancements will thus require modifications to the ATAMM Simulator as well as other tools developed around the ATAMM. In addition, any enhancements or modifications to the ATAMM model will no doubt spawn other meaningful research topics for future consideration.

REFERENCES

[1] Sukhamoy Som, "Performance Modeling and Enhancement for the ATAMM Data Flow Architectures," Ph. D. Dissertation, Old Dominion University, Norfolk, Virginia, May 1989.

[2] T. Agerwala and Arvind, "Data Flow Systems," *Computer*, pp. 10-13, February 1982.

[3] J. Tiberghien, *New Computer Architectures*, Academic Press, London, 1984.

[4] Tadao Murata, "Relevance of Network Theory to Models of Distributed/Parallel Processing," *Journal of Franklin Institute*, pp. 41-49, 1980.

[5] Tadao Murata, "Synthesis of Decision-Free Concurrent Systems for Prescribed Resources and Performance," *IEEE Transactions on Software Engineering*, pp. 525-530, November 1980.

[6] J. W. Stoughton and R. R. Mielke, "Petri-Net Model for Concurrent Processing of Complex Algorithms," *Proceedings of Government Microcircuit Applications Conference*, San Diego, CA, November 1986.

[7] R. R. Mielke, J. W. Stoughton, And S. Som, "Modeling and Performance Bounds for Concurrent Processing," *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, CA, June 1988.

[8] Roland R. Mielke, John W. Stoughton and Sukhamoy Som, "Modeling and Optimum Time Performance for Concurrent Processing," NASA Contractor Report 4167, August 1988.

[9] C. M. Krishna, K. G. Shin, and I. S. Bhandari, "Processor Tradeoffs in Distributed Real Time Systems," *IEEE Transactions on Computers*, vol. 36, pp. 1030-1040, September 1987.

[10] Tadao Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, April 1989.

[11] John W. Stoughton and Roland R. Mielke, "Strategies for Concurrent Processing of Complex Algorithms in Data Driven Architectures," NASA Contractor Report 181657, February 1988.

[12] S. Som, B. Mandala, R. R. Mielke and J. W. Stoughton, "A Design Tool for Computations in Large Grain Real-Time Data Flow Architectures," *Proceedings of the IEEE Southeastcon '90*, New Orleans, Louisiana, April 1990.

[13] S. Som, J. W. Stoughton and R. R. Mielke, "Performance Prediction, Simulation, and Measurement for Real-Time Computing in a Class of Data Flow Architectures," Technical Paper Presented at the ISMM International Conference on Computer Applications in Design, Simulation and Analysis, New Orleans, Louisiana, March 1990.

[14] W. R. Tymchyshyn, "ATAMM Multicomputer System Design," Master's Thesis, Old Dominion University, Norfolk, Virginia, August 1988.

[15] R. R. Mielke, J. W. Stoughton, S. Som, R. Obando, M. Malekpour, and B. Mandala, "Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification," NASA Contractor Report 4339, November 1990.

[16] P. J. Hayes, R. L. Jones, H. F. Benz, A. M. Andrews, J. W. Stoughton, R. R. Mielke, M. Malekpour, and P. R. Appleget, "VHSIC Multiprocessor Implementation of the ATAMM Strategy," *GOMAC91/ 1991 Digest of Papers*, pp. 521-525, November 1991.

[17] Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publication Company Incorporated, 1987 pp. 2.

[18] R. L. Jones, "Diagnostics Software for Concurrent Processing Computer Systems," Master's thesis, Old Dominion University, Norfolk, VA, August 1990.

[19] S. R. Ladd, "Performance Issues," *Computer Languages*, pp. 125-128, August 1989.

[20] M. Malekpour, R. Obando, R. R. Mielke, and J. W. Stoughton, "ATAMM Simulation Tool for Data Flow Architectures," *Proceedings of the 21st Annual Pittsburgh Conference on Modeling and Simulation*, May 1990.

[21] B. Mandala, "A Software Design Tool for Predictable Performance in Real-Time, Data Flow Architectures," Master's thesis, Old Dominion University, Norfolk, VA, December 1990.

APPENDICES

A.1 Overview

The data structure of AMOS is tailored for the use of the Simulator development. Section A.2 is a detailed description of this data structure. Some examples of the input and output files of the Simulator are listed in Section A.3. Figure A.1 is an example of an GPH file. Figure A.2 is an example of an STP file and Figure A.3 is an example of an CTL file.

A.2 Description of Simulator Data Structure

The data structure of AMOS consists of two arrays, BLOCKS and EDGES, that hold all of the information regarding nodes and edges of an algorithm graph. Also, there is a table, PRIORITY, that holds information regarding order of precedence of nodes of the algorithm graph. In addition, there are four queues, QUEUE, WORK, DIAG, and RECOV, that hold information about current status of functional units. The QUEUE is an FIFO queue of functional units, WORK is a pool of working functional units, DIAG is diagnostics pool, and RECOV is a pool of functional units to be recovered by the system. In this section a detailed description of this data structure is presented.

Every functional unit, every 1750A, has an instance of AMOS. After every F, D, and R command, the AMOS structure is updated by the 1750A and broadcast to all other active 1750As. To make the broadcasting more manageable the variables BLOCKS, EDGES, PRIORITY, QUEUE, and etc. are defined as arrays. Also these arrays are assumed to reside in memory back to back so that broadcasting is accomplished by simply copying a portion of memory of one 1750A to all other 1750As. Although these variables are defined as arrays, they are treated as link lists, i.e., the link list is

implemented using array indices. The link list structure reflects the dynamic structure inherent in this architecture model.

FIRING is a global variable that holds the ID of the block being fired. It is used to ensure that all of the colored-nodes of the block are fired before firing the next block. A block is a node of AMG. In TMR mode, it is a set of three colored-nodes, red, green, and blue and in SIMPLEX mode a set of only one node. Its primary use is in TMR mode. If there is no block being fired, then it is set to zero. MODE, a global variable, indicates the mode of operation and is initially set by the user to SIMPLEX, 1, or TMR, 3. In TMR mode, when the number of functional units drops to less than three, AMOS will change the value of MODE to SIMPLEX to reflect the decrease in the number of functioning resources. BLOCKS is an array of N elements with components BLOCKS[j], the range of j is from 0 to N, where N represents the number of nodes in the AMG graph. EDGES is an array of M elements with components EDGES[k], the range of k is from 0 to M, where M represents total number of edges in the AMG graph. QUEUE, WORK, DIAG, and RECOV are arrays of size equal to the maximum number of available functional units at the start up. These arrays are described in the following paragraphs.

BLOCKS

BLOCKS[j] is an element of the array BLOCKS and holds all information about a block. BLOCKS[j] consists of nine variables. FUNCTION_ID is an integer representing the task ID or a pointer pointing to the application program. ID is a three element array which holds ID of functional units assigned to the colored-nodes of the block. It is used to keep track of functional units for future recovery purposes. BUSY_CTR is a counter that holds the number of functional units working on the block. It is incremented after every F-transition command and decremented after every D-transition command. DONE_CTR is a counter that holds the number of functional units

released from the block. It is used to check if a block can be enabled. It is set to zero when the block is enabled and is incremented by every D-transition. ENABLE_CTR is a counter that holds the number of enabled colored-nodes that have not yet fired. When the block is firable the ENABLE_CTR is set to the MODE of operation. It is decremented after every colored-node of a block is fired (F-transition). INPUTS is an array of pointers having components INPUTS[i], the range of i is from 0 to 2. INPUTS[i] is the header pointer pointing to a link list of input (incoming data) edges to the ith colored-node. OUTPUTS is an array of pointers having components OUTPUTS[i], the range of i is from 0 to 2. OUTPUTS[i] is the header pointer pointing to a link list of output (outgoing data) edges originating from the ith colored-node. (It implicitly represents all backward control edges from all successor nodes to this node.) Figure A.4 is a pictorial representation of these two link lists. IN_SUMMARY is an array of integers with components IN_SUMMARY[i], the range of i is from 0 to 2. IN_SUMMARY[i] is a summary of INPUTS[i]. It is an integer having a value equal to the number of input edges of the ith colored-node when all have data and is zero otherwise. OUT_SUMMARY is an array of integers with components OUT_SUMMARY[i], the range of i is from 0 to 2. OUT_SUMMARY[i] is a summary of OUTPUTS[i]. It is an integer having value equal to the number of outgoing edges originating from the ith colored-node when all are empty and is zero otherwise. A block is enabled under the following conditions:

1. DONE_CTR = MODE,
2. All IN_SUMMARY[i]s, i = 0..2, are non-zero, and
3. All OUT_SUMMARY[i]s, i = 0..2, are non-zero.

EDGES

EDGES[k] is an element of the array EDGES and holds all information about an edge. EDGES[k] consists of eleven variables. EDGE_QUEUE is a circular link list that

holds addresses of the memory locations where the data are stored. The addresses are accessible to the INITIAL and TERMINAL blocks to write and read data, respectively. For future recovery purposes the length of the queue, L, is one more than the SEGMENTS or, number of Dummy nodes plus two. Structure of each element of the EDGE_QUEUE consists of three elements; a) LABEL is a pointer to the beginning of the data container, b) ID holds the ID of the functional unit which wrote the data into that data container, and c) NEXT is a pointer to the next element of the EDGE_QUEUE. SEGMENTS is an integer equal to the number of dummy nodes on the edge plus one. It is used to check capacity of the EDGE_QUEUE of the edge. If SEGMENTS is equal to ITEMS, then EDGE_QUEUE is full and no more data can be written into it. ITEMS is a counter indicating the number of data items on the edge. The range of ITEMS is from zero to SEGMENTS. It is incremented, by the INITIAL node, every time new data are written on the edge. It is decremented, by the TERMINAL node, every time OUTPUT_WIDTH becomes zero. INITIAL holds the block number of the origin of the edge. It is used to update the graph and can also be used to check the integrity of the graph. TERMINAL holds the block number of the destination of the edge. It is used to update the graph. EDGE_COLOR indicates the color of the INITIAL node of the edge. It is also used to update the graph. The value of color is identified as 1 for red, 2 for green, and 3 for blue. OUTPUT_WIDTH, a counter, is set to MODE when its present value is zero and ITEMS is non-zero. It is decremented by one for each F-transition of the TERMINAL block. TERMINAL_PTR is a pointer to the element of the EDGE_QUEUE where the TERMINAL node reads data. It is updated every time OUTPUT_WIDTH becomes zero. Updating TERMINAL_PTR means that it should be pointing to the next element of the EDGE_QUEUE. Updating is performed by the TERMINAL node. INITIAL_PTR is a pointer to the element of the EDGE_QUEUE where the INITIAL node writes data. It is updated every time an output is written to the edge. Updating INITIAL_PTR means that it should be pointing to the next element of

the EDGE_QUEUE. Updating is performed by the INITIAL node. NEXT_INPUT is a pointer to the next edge which is an input edge to the TERMINAL block. NEXT_OUTPUT is a pointer to the next edge which is an output edge of the INITIAL block. NEXT_INPUT and NEXT_OUTPUT are used to examine all of the input and output edges of a block, respectively.

QUEUE

Each element of the QUEUE is a record of three components ID, COLOR, and NEXT. ID holds ID of an available functional unit. COLOR is a variable containing the color of the colored-node of the enabled block that the functional unit will process. COLOR carries valuable information only when it belongs to one of the top MODE elements of the QUEUE. The COLOR value is assigned according to the position of the functional unit in the top of the QUEUE; first red, second green, and third blue. NEXT holds the index of the next element of the QUEUE. It is used to treat this array as a link list. If NEXT is zero, then there are no more elements in the list. The first element of this array is used as dummy head node of the link list and to keep track of content of the array, more specifically, COLOR field of the first element holds the number of functional units in the array.

WORK, DIAG, RECOV

WORK, DIAG, and RECOV have the same structure as QUEUE but are treated differently. QUEUE is a FIFO queue while WORK, DIAG, and RECOV are pools of functional units. WORK is a pool holding ID of all of the working functional units. DIAG is a pool holding ID of all of the in-test functional units. RECOV is also a pool of functional units but it holds ID of the resources to be recovered by the system.

PRIORITY

It is an array holding block numbers. The position in the array determines the block's priority. The block at the first element is the block with the highest priority in the graph.

A.3 Figures

```
..Simulator.Version.2.0..
NODES   6
INDEX   1
ID      1
NEXT    0
ENB_CTR  0
BSY_CTR  0
DON_CTR  0
READ_TIME 0
PROS_TIME 60
WRTE_TIME 3
INPUTS   5 25 45
OUTPUTS  10 30 50
LOCATION  125 143 165 183
INDEX   2
ID      2
NEXT    0
ENB_CTR  0
BSY_CTR  0
DON_CTR  0
READ_TIME 0
PROS_TIME 310
WRTE_TIME 3
INPUTS   1 21 41
OUTPUTS  2 22 42
LOCATION  120 292 160 332
INDEX   3
ID      3
NEXT    0
ENB_CTR  0
BSY_CTR  0
DON_CTR  0
READ_TIME 0
PROS_TIME 70
WRTE_TIME 3
INPUTS   6 26 46
OUTPUTS  9 29 49
LOCATION  228 222 268 262
INDEX   4
ID      4
NEXT    0
ENB_CTR  0
BSY_CTR  0
```

```
DON_CTR  0
READ_TIME 0
PROS_TIME 1240
WRTE_TIME 3
INPUTS   7 27 47
OUTPUTS  8 28 48
LOCATION  337 143 377 183
INDEX    5
ID       5
NEXT     0
ENB_CTR  0
BSY_CTR  0
DON_CTR  0
READ_TIME 0
PROS_TIME 100
WRTE_TIME 3
INPUTS   2 22 42
OUTPUTS  3 23 43
LOCATION  326 302 366 342
INDEX    6
ID       6
NEXT     0
ENB_CTR  0
BSY_CTR  0
DON_CTR  0
READ_TIME 0
PROS_TIME 1050
WRTE_TIME 3
INPUTS   10 30 50
OUTPUTS  4 24 44
LOCATION  422 234 462 274
.....
EDGES    30
INDEX    1
ID       1
NEXT     0
KIND     0
TOKEN    0
SEGMENT  1
EDG_COLOR 1
ITEMS    0
OUT_WIDTH 0
NEXT_IN  0
NEXT_OUT 0
TERMINAL  2
INITIAL  1
```

TERM_KIND 1
INIT_KIND 0
LINE_SEG 1
POINTS 88 233 120 312
LOCATION 88 233 120 312
Q_LOC 94 262 114 282
INDEX 2
ID 2
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 5
INITIAL 2
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 160 312 326 322
LOCATION 160 312 326 322
Q_LOC 233 307 253 327
INDEX 3
ID 3
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 6
INITIAL 5
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 366 322 422 254
LOCATION 366 254 422 322
Q_LOC 384 278 404 298
INDEX 4
ID 4

NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 1
INITIAL 6
TERM_KIND 3
INIT_KIND 1
LINE_SEG 1
POINTS 462 254 523 259
LOCATION 462 254 523 259
Q_LOC 482 246 502 266
INDEX 5
ID 5
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 1
TERMINAL 1
INITIAL 1
TERM_KIND 1
INIT_KIND 0
LINE_SEG 1
POINTS 88 233 125 163
LOCATION 88 163 125 233
Q_LOC 96 188 116 208
INDEX 6
ID 6
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0

NEXT_OUT 0
TERMINAL 3
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 165 163 228 242
LOCATION 165 163 228 242
Q_LOC 186 192 206 212
INDEX 7
ID 7
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 6
TERMINAL 4
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 165 163 337 163
LOCATION 165 163 337 163
Q_LOC 241 153 261 173
INDEX 8
ID 8
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 3
NEXT_OUT 0
TERMINAL 6
INITIAL 4
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 377 163 422 254
LOCATION 377 163 422 254

Q_LOC 389 198 409 218
INDEX 9
ID 9
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 8
NEXT_OUT 0
TERMINAL 6
INITIAL 3
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 268 242 422 254
LOCATION 268 242 422 254
Q_LOC 335 238 355 258
INDEX 10
ID 10
NEXT 0
KIND 0
TOKEN 0
SEGMENT 2
EDG_COLOR 1
ITEMS 0
OUT_WIDTH 0
NEXT_IN 9
NEXT_OUT 7
TERMINAL 6
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 165 163 422 254
LOCATION 165 163 422 254
Q_LOC 283 198 303 218
INDEX 21
ID 21
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2

ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 2
INITIAL 1
TERM_KIND 1
INIT_KIND 0
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 22
ID 22
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 5
INITIAL 2
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 23
ID 23
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 6
INITIAL 5
TERM_KIND 1
INIT_KIND 1

LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 24
ID 24
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 1
INITIAL 6
TERM_KIND 3
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 25
ID 25
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 21
TERMINAL 1
INITIAL 1
TERM_KIND 1
INIT_KIND 0
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 26
ID 26
NEXT 0
KIND 0

TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 3
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 27
ID 27
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 26
TERMINAL 4
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 28
ID 28
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 23
NEXT_OUT 0
TERMINAL 6

INITIAL 4
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 29
ID 29
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 28
NEXT_OUT 0
TERMINAL 6
INITIAL 3
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 30
ID 30
NEXT 0
KIND 0
TOKEN 0
SEGMENT 2
EDG_COLOR 2
ITEMS 0
OUT_WIDTH 0
NEXT_IN 29
NEXT_OUT 27
TERMINAL 6
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 41

ID 41
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 2
INITIAL 1
TERM_KIND 1
INIT_KIND 0
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 42
ID 42
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 5
INITIAL 2
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 43
ID 43
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0

NEXT_IN 0
NEXT_OUT 0
TERMINAL 6
INITIAL 5
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 44
ID 44
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 1
INITIAL 6
TERM_KIND 3
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 45
ID 45
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 41
TERMINAL 1
INITIAL 1
TERM_KIND 1
INIT_KIND 0
LINE_SEG 1
POINTS 0 0 0 0

LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 46
ID 46
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 0
TERMINAL 3
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 47
ID 47
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 0
NEXT_OUT 46
TERMINAL 4
INITIAL 1
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 48
ID 48
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1

EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 43
NEXT_OUT 0
TERMINAL 6
INITIAL 4
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 49
ID 49
NEXT 0
KIND 0
TOKEN 0
SEGMENT 1
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 48
NEXT_OUT 0
TERMINAL 6
INITIAL 3
TERM_KIND 1
INIT_KIND 1
LINE_SEG 1
POINTS 0 0 0 0
LOCATION 0 0 0 0
Q_LOC 0 0 0 0
INDEX 50
ID 50
NEXT 0
KIND 0
TOKEN 0
SEGMENT 2
EDG_COLOR 3
ITEMS 0
OUT_WIDTH 0
NEXT_IN 49
NEXT_OUT 47
TERMINAL 6
INITIAL 1
TERM_KIND 1

```
INIT_KIND 1
LINE_SEG  1
POINTS    0 0 0 0
LOCATION  0 0 0 0
Q_LOC     0 0 0 0
.....
SOURCES   1
INDEX     1
ID        1
NEXT      0
ENB_CTR   0
BSY_CTR   0
DON_CTR   0
WRTE_TIME 3
PROS_TIME 0
INJT_TIME 1265
OUTPUTS   5 25 45
LOCATION  8 223 88 243
.....
SINKS     1
INDEX     1
ID        1
NEXT      0
ENB_CTR   0
BSY_CTR   0
DON_CTR   0
READ_TIME 0
INPUTS    4 24 44
LOCATION  523 249 583 269
```

Figure A.1. GPH file, Space Surveillance Algorithm.

```
MODE 1
NumofFUS    4
Protocol    1
Grab_Time   1
BDCT_Time   1
Test_Time   3
Updt_Time   0
Wait_Time   0
```

Figure A.2. STP file, an example.

1265 1265 1265 1265 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0
2 0 0 0
3 0 0 0
4 0 0 0
5 0 0 0
6 0 0 0
7 0 0 2
8 0 0 3
9 0 0 0
10 0 0 0
4
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
2 4 3 1
2 3 2 1
4 1 2 2
4 4 6 2
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Figure A.3. The CTL file for the Space Surveillance Algorithm.

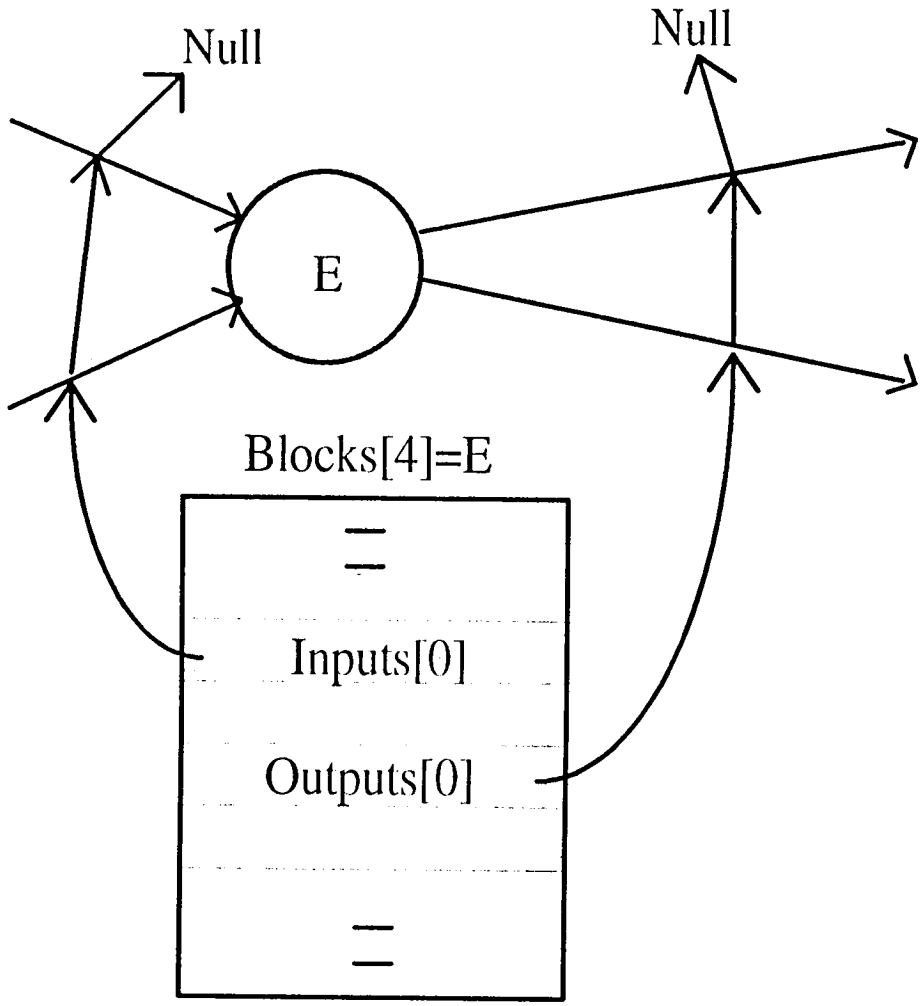

Figure A.4. A Pictorial Representation of Inputs and Outputs Link Lists.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	Contractor Report 3/1/90-2/28/91

4. TITLE AND SUBTITLE

Simulator for Concurrent Processing Data Flow Architectures

5. FUNDING NUMBERS

G NCC1-136

590-32-31-01

6. AUTHOR(S)

Mahyar R. Malekpour, John W. Stoughton, and Roland R. Mielke

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Old Dominion University Research Foundation
P. O. Box 6369
Norfolk, Virginia 23508-0369

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR-189604

11. SUPPLEMENTARY NOTES

Langley Technical Monitor: Paul J. Hayes

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited

Subject Category 33

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A software simulator capable of simulating execution of an algorithm graph on a given system under the Algorithm to Architecture Mapping Model (ATAMM) rules is presented. ATAMM is capable of modeling the execution of large-grained algorithms on distributed data flow architectures. Investigating the behavior and determining the performance of an ATAMM based system requires the aid of software tools. The ATAMM Simulator presented is capable of determining the performance of a system without having to build a hardware prototype. Case studies are performed on four algorithms to demonstrate the capabilities of the ATAMM Simulator. Simulated results are shown to be comparable to the experimental results of the Advanced Development Model system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Simulation software; Dataflow architecture; Multicomputer operating system; Petri nets; Concurrent processing	137
	16. PRICE CODE A07

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL